Techniques for Efficient Regular Expression Matching Across Hardware Architectures

A Thesis

presented to

the Faculty of the Graduate School

at the University of Missouri-Columbia

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

XIANG WANG

Dr. Michela Becchi, Thesis Supervisor

July 2014

The undersigned, appointed by the dean of the Graduate School, have examined the thesis entitled

TECHNIQUES FOR EFFICIENT REGULAR EXPRESSION MATCHING ACROSS HARDWARE ARCHITECTURES

presented by Xiang Wang,

a candidate for the degree of master of science,

and hereby certify that, in their opinion, it is worthy of acceptance.

Professor Michela Becchi

Professor Harry Tyrer

Professor Guilherme DeSouza

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my research advisor Michela Becchi for her advising and teaching during my graduate study. Her enthusiasm and dedication inspired me to overcome difficulties and realize new ideas. I am extremely grateful for her instructions not only on academic research but also on normal life. I also learned how to share, respect and become a better person from her.

I have been so lucky to work with my friends in the NPS lab. In particular, I would like to thank all the lab members, including Kittisak, Daniel, Henry, Ruidong and etc, for the knowledge they shared and the happiness they provided.

I have been so fortunate to have summer intern at Micron Technology. It was great pleasure to work with knowledgeable people there. I am thankful for their help on both my work and life. I enjoyed this experience and benefited a lot from it.

I am thankful to my nice friends for bringing me the beautiful life here in Columbia during these years. Especially, I will always remember the everlasting friendship with Yuanhao, Lei, Yifan, Yi, Xiangjie, Sangdi, Haidong, Xiao, Xiuyi, Muhan and Luyao.

Finally, I am extremely grateful for having a nice family. With their encouragement and support, I became more confident and optimistic about my study and life. In particular, I have great parents who taught me with patience and help me to solve challenging problems.

Thank you!

Xiang

University of Missouri-Columbia,

July 2014

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

TECHNIQUES FOR EFFICIENT REGULAR EXPRESSION MATCHING ACROSS HARDWARE ARCHITECTURES

Xiang Wang

Dr. Michela Becchi, Thesis Supervisor

ABSTRACT

Regular expression matching is a central task for many networking and bioinformatics applications. For example, network intrusion detection systems, which perform deep packet inspection to detect malicious network activities, often encode signatures of malicious traffic through regular expressions. Similarly, several bioinformatics applications perform regular expression matching to find common patterns, called motifs, across multiple gene or protein sequences. Hardware implementations of regular expression matching engines fall into two categories: memory-based and logic-based solutions. In both cases, the design aims to maximize the processing throughput and minimize the resources requirements, either in terms of memory or of logic cells.

Graphical Processing Units (GPUs) offer a highly parallel platform for memory-based implementations, while Field Programmable Gate Arrays (FPGAs) support reconfigurable, logic-based solutions. In addition, Micron Technology has recently announced its Automata Processor, a memory-based, reprogrammable hardware device. From an algorithmic standpoint, regular expression matching engines are based on finite automata, either in their non-deterministic or in their deterministic form (NFA and DFA, respectively). Micron's Automata Processor is based on a proprietary Automata Network, which extends classical NFA with counters and boolean elements.

In this work, we aim to implement highly parallel memory-based and logic-based regular expression matching solutions. Our contributions are summarized as follows. First, we implemented regular expression matching on GPU. In this process, we explored compression techniques and regular expression clustering algorithms to alleviate the memory pressure of DFA-based GPU implementations. Second, we developed a parser for Automata Networks defined through Micron's Automata Network Markup Language (ANML), a XML-based high-level language designed to program the Automata Processor. Specifically, our ANML parser first maps the Automata Networks to an internal representation. We then apply NFA optimization techniques designed for other architectures to this internal representation. Finally, we implemented a tool to convert our internal representation to Verilog, thus allowing automatic deployment on FPGA. Our toolchain allows the user to apply existing optimization techniques to Micron's Automata Processor and to directly compare this new platform with FPGA-based solutions.

Chapter1 Introduction

Pattern matching is at the center of many applications in a variety of domains. For example, deep packet inspection in network security and genome sequence search in the bioinformatics area highly rely on pattern matching. To detect malicious network activities and avoid network intrusion, most networking applications perform regular expression matching on packets. To discover specified gene sequences which may cause particular diseases, the searches of common gene sequences are conducted for different sequence samples.

Regular expression matching has been traditionally performed using finite automata, either in their deterministic or in their non-deterministic form (DFA and NFA, respectively). These data structures reduce the search process to a basic graph traversal guided by the symbols in the input stream. Both the NFA and DFA data structures can be deployed on different hardware platforms. Network processors, ASICs (Application-Specific Integrated Circuits) and FPGAs are widely used in network devices for packet inspection. Due to their massive computational power, GPGPUs (General Purpose GPUs) have proven to be a viable candidate for regular expression matching on large datasets. In addition, Micron Technology has recently announced their Automata Processor, a memory-based accelerator of NFA-wise computations.

In this thesis, we study the deployment of regular expression matching on three hardware platforms: GPUs, FPGAs and Micron’s Automata Processor. In particular, we consider the effective deployment of DFA-based search engines on GPU, and of NFA-based search engines on FPGA and on the Automata Processor.

1.1 Contributions

In this thesis, we discuss regular expression matching on different hardware platforms and propose algorithms to further optimize these implementations. Our main contributions can be summarized as follows.

First, we implement a compression algorithm for DFA to fit the limited GPU memory resources. We consider different implementation alternatives that are suited to the GPU architecture and its memory hierarchy. Furthermore, we propose two regular expression clustering algorithms that allow generating relative compact DFAs to fit the GPU memory even in the presence of complex pattern-sets.

Second, we develop a programming interface for Micron's Automata Processor. Micron's Automata Processor accelerates the traversal of so-called Automata Networks, which are extensions to NFA with counter and boolean element. Those Automata Networks can be represented through an XML-based language called ANML (Automata Network Markup Language). We develop an ANML generator and parser. The former allows deploying our optimized NFA onto the Automata Processor. The latter allows importing existing ANML specification into C++ data structures for optimizations (such as NFA reduction, alphabet reduction and so on).

Finally, we develop an automatic Verilog generator to transfer Automata Networks to Verilog files that target FPGA implementation.

1.2 Organization

The remainder of this thesis is organized as follows. In Chapter 2, we provide background on finite automata and the hardware platforms we used in our research. In Chapter 3, we describe the design of our DFA-based search engine targeting GPU. We also propose a DFA compression algorithm and two regular expressions clustering algorithms. In Chapter 4, we present the implementation of our ANML parser and ANML generator as useful tools for both Micron's Automata Processor and FPGA. In Chapter 5, we discuss the deployment of Automata Networks on FPGAs by developing a generator to convert NFA or Automata Networks to Verilog files. Finally, in Chapter 6, we summarize our main results and discuss several future research directions.

Chapter 2 Background

In this chapter, we provide background on regular expressions matching. In Section 2.1, we discuss regular expression. In Section 2.2, we provide an introduction to NFA and DFA. In the remaining sections, we provide background on the considered hardware platforms, namely GPU, FPGA and Micron's Automata Processor.

2.1 Regular Expressions

A regular expression is a sequence of characters and special symbols that represent a set (possible infinite) of exact-match strings. The features found in regular expressions can be classified into different categories.

Exact-match patterns represent a simple string with fixed length. The Aho-Corasick DFA construction algorithm [1] can be used in the case of exact-match patterns.

Character sets accept a set of symbols and can be represented in two ways. First, it can be represented in the form $[c_1\text{-}c_2c_3]$, c_i being any character of the alphabet. For example, *[a-cz]* represents the combination of character *'z'* and a character range that starts from *'a'* and ends at *'c'*. The second way is by special symbols, such as space characters (\s), all digits (\d), all alphanumerical characters (\w), and their complements (\S, \D, \W). In addition, both single characters and character sets can be repeated, using expressions such as $c+$, $c*$, $[c_1c_2]+$ and $[c_1c_2]*$.

Wildcards are represented by a single dot symbol. The wildcard repetition called dot-star (.*) is commonly found in complex regular expressions derived from anti-viruses and network intrusion detection systems and may cause DFA size explosion when large sets of regular expressions are combined.

Finally, counters allow bounded or infinite repetitions of particular characters or patterns. For instance, *a{1,99}* matches a sequence of characters *'a'* ranging from 1 to 99 in length. As analyzed in [3], both counters and wildcards may lead to state blow-up when performing NFA to DFA transformation.

2.2 Finite Automata

Regular expression matching has traditionally been implemented by representing the pattern-set through finite automata (FA) [14], either in their deterministic or in their non-deterministic form (DFA and NFA, respectively). The matching operation is equivalent to a FA traversal guided by the content of the input stream. Worst-case guarantees can be met by bounding the amount of processing performed per character. Being the basic data structure in the regular expression matching engine, the finite automaton must be deployable on a reasonably provisioned hardware platform. As the size of pattern sets and the expressiveness of individual patterns increase, limiting the size of the automaton becomes challenging. The exploration space is characterized by a trade-off between the size of the automaton and the worst-case bound on the amount of per character processing.

NFAs and DFAs are at the two extremes in this exploration space. NFAs [14, 19] have a limited size but can require expensive per-character processing, whereas DFAs offer limited per-character processing (only one state transition is taken for each input character) at the cost of a possibly large automaton.

2.2.1 Introduction to NFA and DFA

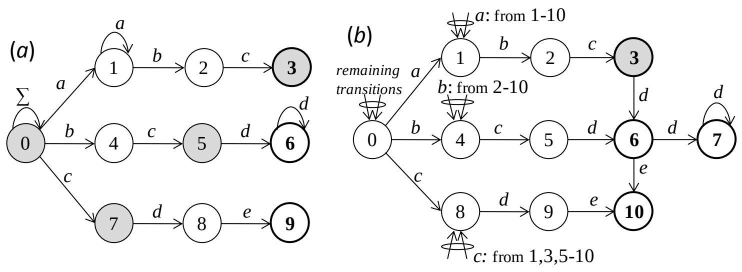


Fig. 2.1: (a) NFA and (b) DFA accepting regular expressions a+bc, bcd+ and cde. Accepting states are bold. States active after processing text aabc are colored gray. In the NFA, Σ represents the whole alphabet. In the DFA, state 4 has an incoming transition on character b from all states except 1 (incoming transitions to states 0, 1 and 8 can be read the same way).

In Figure 2.1, we show the NFA and DFA accepting three simple patterns (*a+bc*, *bcd+* and *cde*). In the two diagrams, states active after processing text *aabc* are colored gray. In the NFA, the number of states and transitions is limited by the number of symbols in the pattern-set. In the DFA, every state presents one transition for each character in the alphabet (Σ). Each DFA state corresponds to a set of NFA states that can be simultaneously active [14]; therefore, the number of states in a DFA equivalent to an N-state NFA can potentially be 2^N. In reality, previous work [3, 7, 17, 24] has shown that this so-called state explosion happens only in the presence of complex patterns (typically those containing bounded and unbounded repetitions of large character sets).

From an implementation perspective, existing regular expression matching engines can be classified as either memory-based [3-4, 7-8, 12, 15-17, 24, 33] or logic-based [6, 9, 23, 27]. For the former, the FA is stored in memory; for the latter, it is stored in combinatorial and sequential logic. Memory-based implementations can be deployed on various platforms (GPUs, network processors, ASICs, FPGAs); logic-based implementations typically target FPGAs. In the latter case, updates in the pattern-set require the underlying platform to be reprogrammed. In a memory-based implementation, the design goals are to minimize the memory size needed to store the automaton and 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). Typically memory-based implementations use a DFA representation, whereas logic-based implementations use an NFA design.

Existing proposals targeting DFA-based, memory-centric solutions have focused on two aspects: (i) designing compression mechanisms to minimize the DFA memory footprint; and (ii) devising novel automata to be used as an alternative to DFAs in case of state explosion. Despite the complexity of their design, memory-centric solutions have three advantages (i) fast reconfigurability, (ii) low power consumption, and (iii) limited flow state; the latter leading to scalability in the number of flows. The one-hot encoding scheme [13] is at the center of all logic-based designs listed above. By encoding each FA state through a flip-flop, this scheme enables easy implementation of NFAs in logic, while limiting the processing time to one clock cycle per input character (both in the average and in the worst case). Unfortunately, by distributing the state information across the FPGA, this solution does not allow easy and efficient context switching between packet flows. In other words, NFA-based, logic-centric solutions allow one to easily achieve peak worst-case performance on a single flow, at the expense of higher power consumption and of a lack of scalability in the number of concurrent flows.

2.3 Introduction to GPUs

GPUs were originally designed for graphic processing. Nowadays, these platforms are considered more general purpose, and regarded as efficient accelerators for a variety of applications. Many scientific applications have been accelerated on NVIDIA GPUs, whose programmability has greatly improved since the advent of the CUDA programming model. We will give a brief introduction to NVIDIA Fermi GPUs [20] that we have used in this research.

2.3.1 General GPU Architecture

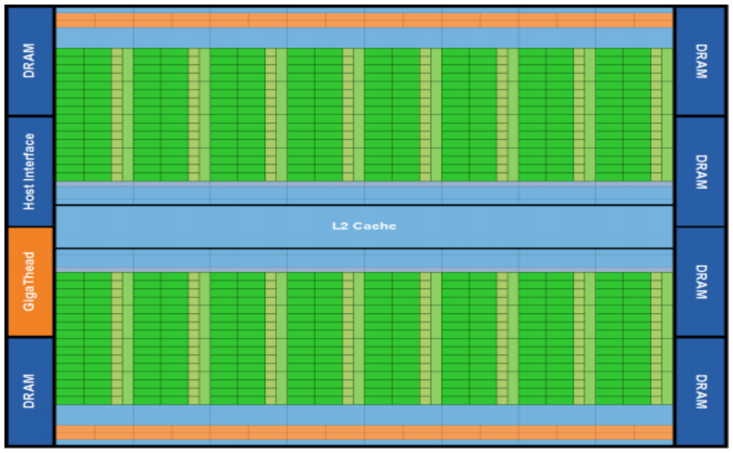


Fig. 2.2: Baseline architecture of Fermi GPU.

In general, GPUs are composed of a number of Streaming Multiprocessors (SMs). Figure 2.2 shows the baseline architecture of Fermi GPU, which consists of 16 SMs and has up to 6 GB of global memory. The host interface connects the GPU to the CPU via a PCI-Express.

In Figure 2.3, we can see the general architecture of a single SM. As can be seen, each SM consists of many simple, in-order cores (usually 32 or 48 for Fermi architecture). 32 threads are grouped into a warp and warps are scheduled on each SM through a warp scheduler. At the same time, a dispatch unit will dispatch different threads to different cores and functional units that will execute the threads’ computation. Threads are also grouped into thread-blocks. Multiple thread-blocks can be mapped to a single SM and multiple threads can be mapped to a single core.

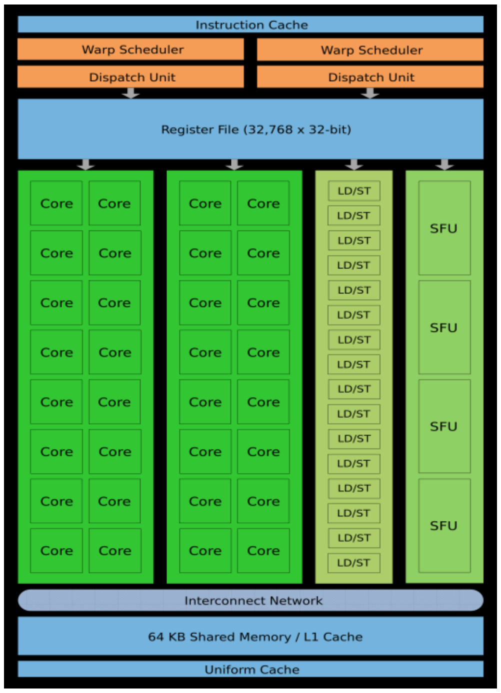


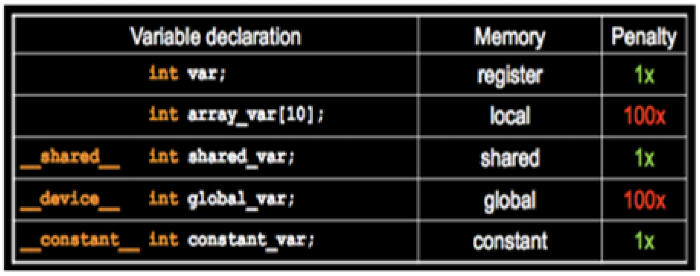
Fig. 2.3: Architecture of streaming multiprocessor on Fermi GPU.

2.3.2 GPU Memory Hierarchy

GPUs have a different memory hierarchy compared to CPU. GPU memories can be classified into 2 categories: on-chip and off-chip memories. First, as for on-chip memory, registers have low access latency and are shared by thread-blocks mapped to the same SM. Second, each SM has a configurable shared memory/L1 cache. Threads within the same block can share the data that reside in the on-chip shared memory on each SM. On the other hand, as a representation of off-chip memory, global memory is the most frequently used one with the largest size and is responsible for the direct communication with host through PCI-e. In addition, constant memory is a cached 64KB read-only memory that belongs to off-chip memory and is shared by all the thread blocks.

As shown in Table 2.1, different kinds of memories provide different access latencies. In general, the GPU memory organization consists of high latency global

Table 2.1: Access latency of GPU memories.



Variable declaration		Memory	Penalty
	int var;	register	1x
	int array_var[10];	local	100x
__shared__	int shared_var;	shared	1x
__device__	int global_var;	global	100x
__constant__	int constant_var;	constant	1x

memory, high latency local memory, low latency read-only constant memory, low-latency read-write shared memory. Therefore, shared memory and constant memory should be preferred when latency is a concern, while global memory needs to be used when memory size requirements become large. Therefore, the judicious use of the memory hierarchy and of the available memory bandwidth is essential to achieve good performance. Detailed considerations about the selection of GPU memories for our DFA-based search engine will be described in Section 3.3.

2.3.3 Threads and blocks on GPUs

Different from other parallel programming models, like POSIX threads [22] and OpenMP [21], CUDA [10] exposes to the programmer two degrees of parallelism: fine-grained parallelism within a thread-block and coarse-grained parallelism across multiple thread-blocks. We can distribute work to a large number of threads by using unique identifiers assigned to threads and blocks. Threads are grouped into warps (32 threads/warp), which operate in a SIMD (Single instruction, multiple data) manner. The configurations of threads and thread-blocks affect the overall performance. Too small configurations cause GPU resource underutilization, while too large configurations result in resource conflicts and in the serialization of parallel computations.

2.4 Introduction to Automata Processor

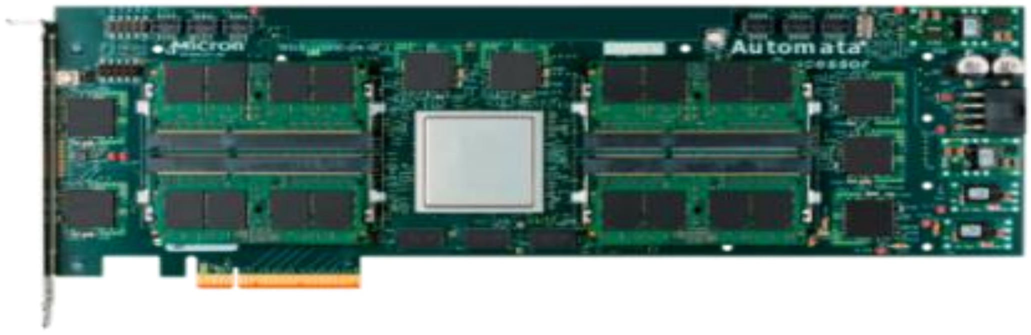


Fig. 2.4: Micron Technology's Automata Processor.

The Automata Processor (AP) [11], as shown in Figure 2.4, is an adoption of SDRAM technology designed to be used as a reconfigurable device for the direct implementation of non-deterministic finite automata (NFA) [11].

2.4.1 General Design

The Automata Processor comprises of an array of thousands of state transition elements (STEs) and a routing matrix. An STE is associated with each state in the mapped automata and stores one state bit that marks if the corresponding state is active or not. Each STE also contains a 256-bit symbol array (indexed by the current input) to process the input symbol. The output of symbol recognition and state bit determine the output of the STE. The output of each STE determines if another STE will become active or inactive after the current input symbol. The next state outputs from all the STEs are connected in parallel to a programmable routing fabric called the "routing matrix". The routing fabric is comprised of an array of switching blocks that allows any STE to communicate with any other STE within a certain physical distance, allowing for a maximum out-degree (fan out) of 16 from most states. Larger out-degrees are possible at the cost of reduced clock rate. Since one STE is associated with each state and is capable of activating another STE, each STE can conceptually be thought of as representing an edge, instead of a state, from the automaton description. The next state tables (and thus edge labels) are stored in the STEs, while the NFA topology is encoded into the configuration of the routing matrix. STEs can be individually marked as being an incoming edge into a start state or an outgoing edge into an accepting state, allowing for multiple start and accepting states. Also, start edges can be further classified as start-of-data or all-input-start, allowing for a simple representation of automata that match substrings.

In general, the Automata Processor consists of 6 ranks, each comprising 8 chips. Patterns can be loaded into the Automata Processor from an object file. Multiple input streams can be scanned in parallel. The Automata Processor thus is able to scan 8~48 flows in parallel and operate at the rate of 128MBps for each rank.

The Automata Processor implements Automata Networks, an extension to NFA, either from PCRE (Perl Compatible Regular Expressions) or from ANML (Automata Network Markup Language). PCRE file can be automatically compiled into a loadable object file by using a compiler provided by Micron Technology. At the same time, ANML, an XML-based language for programming Automata Networks, can be used to construct patterns in the form of XML files that can be further compiled into loadable objects.

2.4.2 Automata Processor Elements

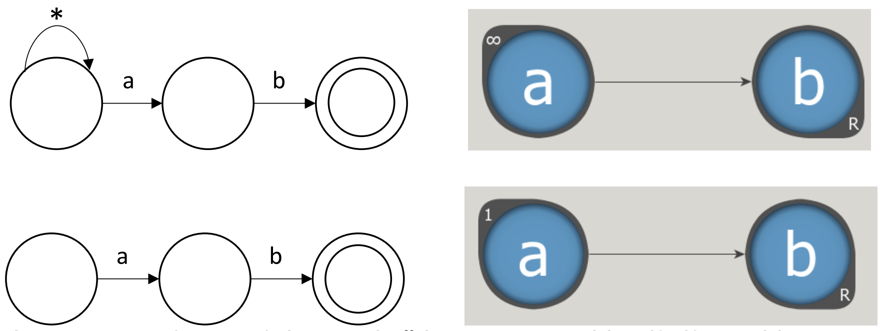


Fig. 2.5: Use of start-of-data and all-input STEs to identify if matching starts at the beginning or anywhere in a given string respectively.

The Automata Processor has 3 basic functional elements: STEs, counters and boolean elements. Each STE represents the normal state in a classical NFA, while counters and boolean elements are used along with STEs to increase the space efficiency of automata implementations and extend the computational capabilities beyond NFA.

A STE can be either configured as start-of-data or all-input to represent a start state as provided in Figure 2.5. In addition, a matching STE has an ‘R’ symbol in the lower right corner. Furthermore, a STE can also be defined as latched which means it will keep being active after once being activated. Finally, unlike classical NFA with labeled transitions, the matching symbols are integrated with the STE (that is, the accepted symbols are associated to the states rather than to the transitions).

Counters allowed some specified patterns to occur for a specified amount of times. Counters include two input ports: count and reset respectively. A target count and counter type should be configured for each counter in advance. In particular, there are three different counter types that can be implemented, namely roll, pulse and latch counters. Like normal counters, the roll counter is reset each time the target value is reached and ready for the next activation. The latch counter persistently activates the elements connected to it on the cycle on which the counter value reaches the target count. The counter value holds at the target and always activates the elements connected to it. The pulse counter activates the elements connected to it on the cycle on which the counter value reaches the target count and on subsequent cycles does not activate the connected elements. The counter value holds at the target but never activate the elements connected to it.

Boolean elements are particularly useful to achieve the functionality of boolean operators, like OR, AND, NOR, NAND, SoP (Sum of products) and PoS (Product of sums).

Table 2.2 shows the resource availability in one chip on the Automata Processor. Each chip consists of two cores and each core has 96 blocks. We have rich STE resource while have limited number of counters and boolean elements in each block. In general, the Automata Network is a powerful extension of classical NFA and therefore especially useful for implementing a variety of pattern matching applications.

Table 2.2: Resource availability in one chip on Automata Processor.

Element	Availability in one chip
STE	49152 in two cores with 96 blocks per core (24576 per core, 256 per block). 6144 can report. (3072 per core, 32 per block)
Counter	768 in two cores (384 per core, 4 per block)
Boolean	2304 in two cores (1152 per core, 12 per block)

2.5 Introduction to FPGAs

An FPGA [18] is a type of integrated circuit (IC) that can be configured to implement a variety of functions in hardware. An FPGA consists of thousands of programmable logic cells shown in Figure 2.6(a). As we can see in this figure, each cell is composed of a look-up table (LUT), a flip-flop and a multiplexer.

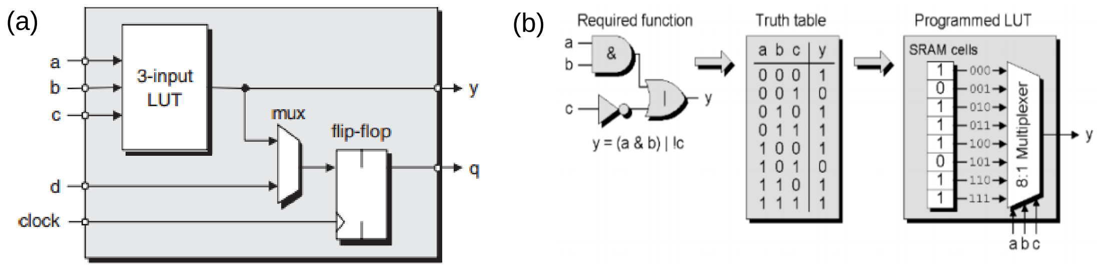


Fig. 2.6: (a) Structure of a logic cell and (b) LUT encoding scheme.

The LUT allows performing logic operations. It can be configured to encode simple logic functions like in Figure 2.6(b). It can also be used as distributed memory. Flip-flops are memory elements storing 1-bit of information. They can be configured to be triggered by a positive- or negative-edge clock. The multiplexer feeding the flip-flop could be configured to accept the output from the LUT or a separate input to the logic block.

In Xilinx FPGAs, a slice consists of multiple programmable logic cells and a configurable logic block (CLB) is made up of multiple slices. As shown in Figure 2.7, fast programmable interconnections also exist between different CLBs.

FPGAs can be programmed using a hardware description language (HDL), like Verilog and VHDL. By making use of these languages, we can easily construct the logic functions that we need to implement. HDL specification can then be processed by synthesis, map, place and route. The outcome of this process is a bit file that can be loaded on the FPGA thus programming its hardware.

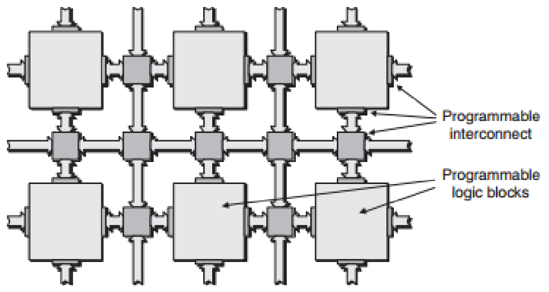


Fig. 2.7: Interconnections between CLBs.

Chapter 3 Regular Expression Matching On GPU

Since DFAs have the potential of state explosion for complex regular expressions, generating DFA representations without exceeding the GPU memory size becomes an important problem. We propose techniques to reduce the DFA size and optimize regular expression matching on GPU. This chapter is structured as follows.

In Section 3.1, we show the basic approach to implement regular expression matching on GPU. In Section 3.2, we discuss an optimization algorithm called alphabet reduction. We propose different approaches to implement alphabet reduction on GPU in Section 3.3 and 3.4. In Section 3.5, we describe two novel clustering algorithms for regular expressions. Our algorithms allow achieving smaller number of DFAs that fit GPU memory. Finally, in Section 3.6 we perform experiments on both real and synthetic pattern-sets using our DFA-based search engine.

3.1 Basic Implementation on GPU

Since GPUs are memory-based hardware platforms. In this case, DFA representation is preferred. DFAs need to be stored in GPU global memory. We take advantage of the uncompressed DFA-based solution from [34]. In general, different threads process different DFAs in the same thread-block. Different input streams are again mapped onto different thread-blocks. Because the ASCII table size is 256, we will store 256 transitions for each state in the memory layout. However, if the regular expression is complex and causes the DFA to have large number of states, the memory requirement may exceed the

GPU memory capacity. In this case, we can apply the alphabet reduction algorithm described in Section 3.2 to reduce the size of transition table and we can implement the clustering algorithms discussed in Section 3.5 to divide the regular expressions into partitions and generate smaller number DFAs that fit memory size.

3.2 Alphabet Reduction

```
procedure alphabet_reduction (DFA $dfa=(n, \delta(states, \Sigma))$, modifies set class);
(1) int alphabet_size = 0;
(2)   for state $s \in states$ do
(3)     for state $t \in states$ do
(4)       set char_covered[$|\Sigma|$] = false;
(5)       set class_covered[$|\Sigma|$] = false;
(6)       set remap[$|\Sigma|$] = 0;
(7)       for (char $c \in \Sigma$ & $\delta(s,c)=t$) do
(8)         char_covered[c] = true;
(9)         class_covered[class[c]]=true;
(10)      for (char $c \in \Sigma$) do
(11)        if (!char_covered [c] & class_covered[class[c]]) then
(12)          if (remap[class[c]]==0) then remap[class[c]]= ++alphabet_size;
(13)          class[c]=remap[class[c]];
end;
```

The idea at the basis of alphabet reduction is the following: in a DFA recognizing regular expressions over an alphabet Σ, each state has potentially $|\Sigma|$ outgoing transitions, one for each symbol in Σ. However, Σ can be partitioned into classes of symbols $C_1,..,C_k$ which are indistinguishable for the purposes of the DFA operation. Two symbols c_i and c_j will fall into the same class if they are treated the same way in all DFA states. In other words, given the transition function $\delta(states, \Sigma) \rightarrow states$, $\delta(s,c_i)= \delta(s,c_j)$ for each state s in the DFA. Once the class translation $C(\Sigma)\rightarrow \{1..k\}$ has been computed, the alphabet is reduced from cardinality $|\Sigma|$ to k. k next state transitions will therefore suffice at each state. An additional alphabet translation table encoding the symbol-to-class mapping is required to allow the pattern matching operation. In practical scenarios (ASCII alphabet) this table will contain 256 entries, with a maximum width of 1 byte (for heavily compressed alphabets 5-6 bits per character may suffice). This indexing table can be efficiently cached or stored in on-chip memory. The algorithm of alphabet reduction is shown in the pseudo code above.

To compute the required alphabet translation tables, we use a parallel variant of the alphabet compression algorithm proposed in [4], which has O(n^2) time complexity. Specifically, we first construct a separate translation table for each state and then build a global alphabet translation table by progressively merging the state-specific tables from the first phase. On a 8-core processor, our implementation achieves a 4-5x speedup compared to the original single-threaded version [4].

Unfortunately, alphabet reduction becomes less effective as the size of the dataset (and of the corresponding DFA) increases. In fact, on large DFAs it is less likely for different symbols to cause transitions to the same target states. Therefore, in this study we combine alphabet compression with regular expression partitioning. In particular, we propose two new regular expression clustering methods in Section 3.5, and we compare them with the bisection-based scheme proposed in [5].

3.3 Selection of GPU Memory for Alphabet Transition Table

In our implementation, each alphabet translation table consists of 256 1-byte entries, thus requiring 256 B of memory. Below, we discuss advantages and disadvantages of storing the alphabet translation tables in different GPU memories.

- Given its large size (from 1 to about 12 GB depending on the GPU), **global memory** can easily accommodate a large number of alphabet translation tables. The main disadvantage of global memory is its high access latency.
- **Shared memory** offers low access latency at the cost of a limited capacity (from 16KB to 48KB per SM, depending on the configuration). The main limitation of shared memory is the following. Shared memory is SM-specific and has the scope of a single thread-block. If multiple thread-blocks with cumulative shared memory requirements exceeding the available capacity are mapped onto the same SM, their execution is serialized. Thus, storing a large number of alphabet translation tables in shared memory will limit the scalability in the number of packet flows. Specifically, given nAT alphabet translation tables, a shared-memory based implementation can scale up to 48KB/(256B×nAT) concurrent flows, and is more suited to pattern-sets that can be easily compiled into a small number of DFAs.
- **Constant memory** is read-only, has a 64KB size, is shared by all the thread-blocks, offers low access latency, and can be accessed in parallel to shared memory. If every thread in a half-warp requests data from the same address in constant memory, the GPU will generate only a single read request and subsequently broadcast the data to every thread. In addition, constant memory is cached, and therefore consecutive reads to the same address will not lead to any additional memory traffic. However, if the threads in a half-warp require different data, the corresponding 16 reads will be serialized. If multiple, per-DFA alphabet translation tables are used, this memory accesses serialization may impact the performance of our implementation, since different threads process different DFAs.

3.4 Per-DFA vs. Shared Alphabet Translation Tables

In general, alphabet translation tables can be either DFA-specific or shared across multiple DFAs. This design choice involves the following trade-off. Per-DFA alphabet translation tables typically result in smaller alphabets, thus reducing the amount of memory required to store the DFA state transition tables. However, as discussed above, multiple alphabet translation tables limit the flow scalability of shared-memory based implementations, and cause access serialization in constant-memory based implementations. On the other hand, sharing a single alphabet translation table across multiple DFAs generally leads to larger alphabets (it is more likely for a character to be treated differently in distinct DFAs), and thus to larger state transition tables (and memory requirements). As we will discuss in Section 3.6.3, we found the use of a single, shared alphabet translation table to be preferable.

3.5 Regular Expression Clustering Algorithm

In this section, we propose two regular expression clustering schemes aimed to mitigate the state explosion problem [3, 17, 33, 25]. Recall that, in our implementation (Section 3.1), each thread is responsible for the traversal of one DFA, and branch and memory divergence are the main obstacles to achieving high processing throughput. Therefore, when performing regular expression partitioning, it is important to alleviate this performance degradation by limiting the number of DFAs. At the same time, the size of each DFA must be kept small, so to limit the DFA memory requirements to the available GPU capacity. Since we need GPU global memory to store packets, we allocate 80% of global memory to store DFAs.

In order to limit the number of DFAs encoding a particular pattern-set, we need to consider the complexity of each regular expression in that set. Combining many complex patterns in a single DFA can lead to state explosion and prohibitive memory requirements [3, 33]; on the other hand, equally distributing complex regular expressions into multiple DFAs allows limiting the size of each DFA. Smaller DFA have also the benefit of faster generation and compression (for example, alphabet compression has a time complexity which is quadratic in the number of DFA states). Below, we present two schemes to achieve this goal.

3.5.1 Single Set Implementation

As explained in previous work [3, 33], state explosion is linked to the presence of particular sub-patterns in the regular expressions (typically repetitions of wildcards and large character sets). To drive our clustering scheme, we assign a weight to problematic sub-patterns that are frequently found in practical datasets. Specifically, the selected weights depend on the degree of state explosion that each sub-pattern may cause when combining multiple regular expressions into a single DFA. Table 3.1 shows the weights associated to various character set repetitions. Sub-patterns .*, [^\n\r]* and [^\n\r]+ are the most problematic, since they always lead to combinatorial state explosion when combining regular expressions; therefore, they are associated the maximum weight value. Other character set repetitions, such as \w+, \d+ and $[c_1..c_n]+$ (with n<20), may also increase the DFA size: this happens when the repeated character appears in other regular expressions in the same set. Since *\w* represents all alphanumerical characters, the weight

Table 3.1: Weight assigned to different character sets.

Character set	$.^*$	$[\^\backslash n\backslash r]^*$	$[\^\backslash n\backslash r]^+$	$\backslash w^+$
Weight	1	1	1	0.5
Character set	$[c_1..c_n]^*$	$[c_1..c_n]^+$	$\backslash d^+$	
Weight	0.3	0.3	0.2	

associated to *\w+* is larger than that associated to *\d+* (*\d* represents only digits) and *[c_1..c_n]+* (for small n).

Figure 3.1(a) shows the flow diagram of the single set clustering algorithm. As we can see from it, after preprocessing the weights of each regular expression rule, we start from trying to generate a single DFA (*N=1*). We double the number *N* and distribute rules to get *N* rule partitions with even total weights until the total size of *N* DFAs fits the memory requirement (80% of global memory). At the same time, we update the lower bound *L* to be N+1 and upper bound *U* to be 2N each time we need to double the value of N. Then we continue to search for the minimum number of DFAs between the lower bound *L* and upper bound *U*. By using a method similar to binary search, each time we set *N = (L+U)/2* and redefine the value of upper or lower bound. When we reach the edge condition (*N==L* or *N==U*), we will export the smallest number of DFAs that fit memory requirement. In this way, we can find the smallest number of DFAs which satisfy the GPU memory requirement relatively fast.

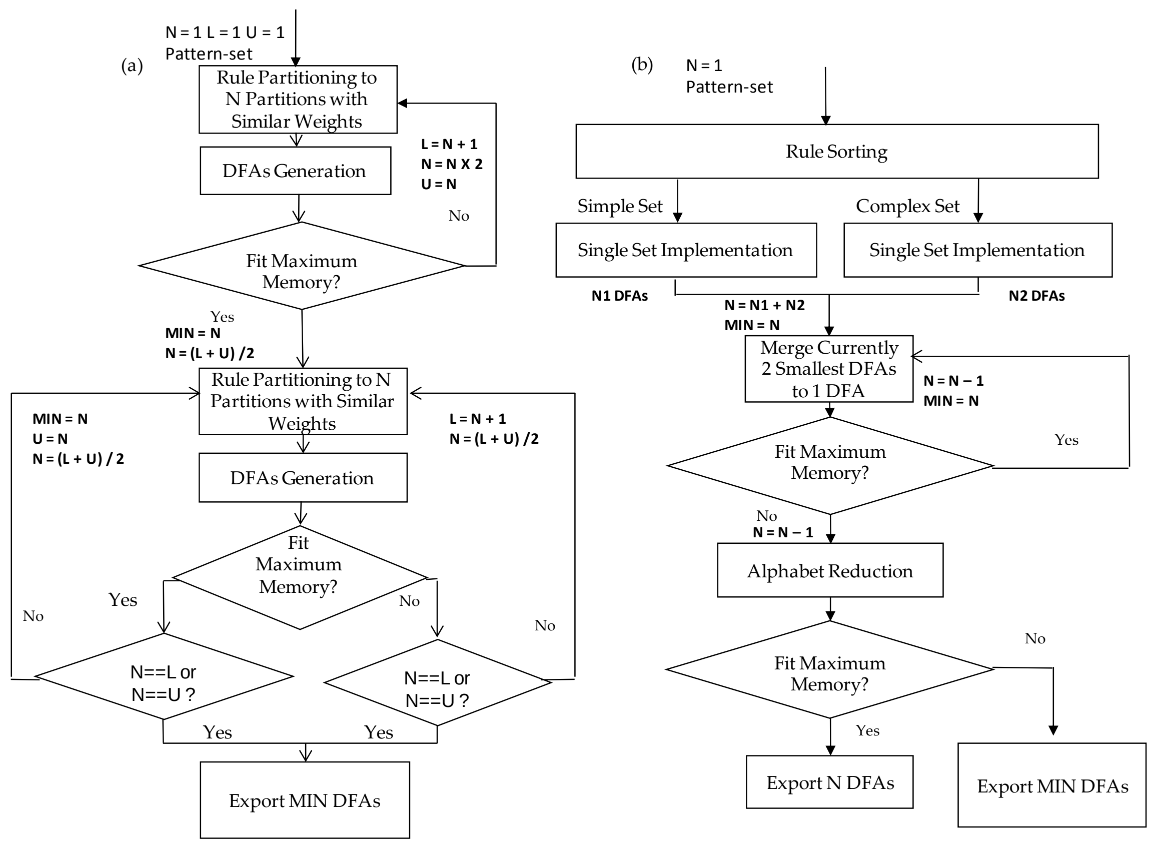


Fig. 3.1: Flow charts of (a) single set implementation and (b) double set implementation.

3.5.2 Double Set Implementation

Single set implementation distributes rules with similar complexity evenly to DFAs in order to get as small number of DFAs as possible. We make the following observations. First, since we want to get a small number of DFAs, we need to gather as many rules, both simple and complex ones, as possible. Second, a complex rule with weight larger than 3 will increase the DFA number of states and potentially causes large DFA. This conflict occurs when we want to gather as many simple rules as possible and at the same time combine them with complex rules. The more simple rules we gather, the larger the DFA size will become and also the larger DFA state replication we will get when combined with complex rules. In this case, large state replication will prevent us from getting relatively small number of DFAs that fit the memory requirement. Therefore, we need to divide relatively simple rules and complex rules into two sets to avoid rapid state increase. We allocate the memory size for each set according to the formulas below:

$$Mem_{complex} = Weight_{complex} / (Weight_{complex} + Weight_{simple}) * Mem_{Total}$$

$$Mem_{simple} = Weight_{simple} / (Weight_{complex} + Weight_{simple}) * Mem_{Total}$$

Since we need GPU global memory to store packets, we define Mem_{total} as 80% the size of global memory and use it to store DFAs generated from both *simple* set and *complex* set. We then allocate the percentage of Mem_{total} to *complex* set according to the ratio of its total weight ($Weight_{complex}$) to the total weight of this pattern-set ($Weight_{complex} + Weight_{simple}$). Similarly, we give the percentage of Mem_{total} to *simple* set according to the ratio of its weight ($Weight_{simple}$) to the total weight of this pattern-set.

The details about double set implementation are shown in Figure 3.1(b). As we can see from it, we divide the rules into two sets named *simple* and *complex* by sorting all the rules. If there are rules with weight larger than the threshold we defined, they are classified as *complex* rules. Otherwise, they belong to the *simple* set. We then implement the single set implementation discussed in Section 3.5.1 for both *simple* and *complex* sets separately to find the minimum number of DFAs. In addition, we define the maximum memory size as $Mem_{complex}$ and Mem_{simple} for *complex* set and *simple* set respectively in the single set implementation. We then merge the smallest two DFAs to one among all the DFAs and update the smallest number (MIN) of DFAs until the total size of DFAs is larger than Mem_{total}. Finally, we perform alphabet reduction. If the total size is not larger than Mem_{total} after alphabet reduction, we export all DFAs into files to be used by GPU implementation. Otherwise, we export the smallest number (MIN) of DFAs that fit Mem_{total} before the last merging. We add merge stage to our new algorithm because of the potential to merge DFAs in these two sets. In this way, we can avoid the case that some DFAs in either set are very small which can be further combined to get smaller total number of DFAs.

3.6 Experimental Evaluation

In this section, we evaluate different alphabet reduction implementations for regular expression matching on GPUs. In addition, we compare the results of our DFA generation algorithms designed for regular expression matching on GPUs. Our experiments are conducted on an 8-core Intel Xeon E5620, running Centos 5.9. The system is equipped with an Nvidia GeForce GTX 480 GPU, comprising 15 32-core SMs. We used CUDA 5.5.

3.6.1 Pattern-sets

We use both real and synthetic pattern-sets in our experiments to evaluate the performances of our system. The real pattern-sets, consisting of *backdoor* and *spyware* rules are drawn from Snort NIDS [26]. These real pattern-sets have various symbol sets (.*, *[^\n\r]** and counters) and up to 7 .* in the most complex patterns. The synthetic pattern-sets are generated by using the tool from [5] and tokens from the Snort rules. We used 4 synthetic pattern-sets called *exact-match (E-M)*, *dotstar0.05*, *dotstar0.1* and *dotstar0.2*. *Exact-match* only has exact match patterns, while dot-star pattern-sets contain

Table 3.2: Weight distribution of regular expressions in pattern-sets.

Dataset	# Regular expressions	<1	[1,2)	[2,3)	[3,4)	[4,6)	>=6
Spyware	462	83	337	12	20	2	8
Backdoor	226	147	63	9	5	2	0
Dotstar0.05	1000	953	44	3	0	0	0
Dotstar0.1	1000	911	78	11	0	0	0
Dotstar0.2	1000	825	151	23	1	0	0

a varying fraction of unbounded repetition of wildcards (5%, 10% and %20 respectively). As for the number of rules, *backdoor* and *spyware* have 226 and 462 rules respectively. All synthetic pattern-sets consist of 1000 rules. In Table 3.2, we show the weight distributions of regular expressions in each of these pattern-sets.

All the packets used in our experiments are generated from the tool described in [5]. By giving 15 probabilistic seeds and 4 traversal probabilities called P_M (35%, 55%, 75% and 95%) which indicate the malicious level of the packets, we generate 15 1-MB trace files for each pattern-set. In addition, we conduct all our experiments by setting the packet size to 64KB.

3.6.2 Effect of GPU Memories

Our first goal is to compare the performances of the three GPU implementations of alphabet reduction described in Section 3.3 (namely implementations using global memory, shared memory and constant memory). Table 3.3 shows the basic characteristics of the DFAs generated for different pattern-sets in our experiments. Since the original alphabet size is 256, by comparing this value with the number of classes after alphabet reduction in column 4 of Table 3.3, we find that the memory size requirement can be saved up to 7.8 times for a single DFA. By comparing column 4 and column 5 in Table 3.3, we can see the merging of multiple alphabet transition tables would increase the

Table 3.3: Characteristics of DFAs.

Dataset	# DFA	# Total states	# Classes (before merging)	# Classes (after merging)
E-M	1	28744	88	88
Backdoor	13	960114	33 ~ 66	110
Spyware	32	95482	18 ~ 51	89
Dotstar.05	16	219330	43 ~ 87	90
Dotstar.1	32	157385	39 ~ 87	90
Dotstar.2	38	1194921	51 ~ 88	90

overall alphabet size. Since characters that belong to the same class in one DFA may lead to different transitions in other DFAs, we need to do further classifications within a single class when merging multiple alphabet transition tables. In Table 3.4, we compare the performances of multiple-table implementation using different GPU memories. We perform our experiments using different number of 64KB packet flows per SM. The use of the optimal number of packet flows per SM (5 in this case) leads to up to 3X speedup over single packet flow per SM. We can first easily discover that the implementation on shared memory achieves the best performance across all the pattern-sets with different trace files because of its low memory access latency. The largest number of DFAs for a pattern-set (*dotstar0.2*) is 38 which require total 9.5KB to store all the tables per block. Therefore, the memory requirements to store alphabet transition tables don't exceed the size of shared memory for all these pattern-sets and shared memory is the ideal location to store alphabet transition tables. Also, the performance by using global memory is between shared memory and constant memory. The relatively high access penalty and low memory coalescing cause the performance of global memory to be not as good as those of shared memory. Finally, we notice the large performance gap between constant memory and the other two memories except for *E-M* pattern-set. The performance loss is

Table 3.4: Throughput (in Mbps) obtained with multiple tables and different memory implementations.

Dataset	$P_M = 0.35$			$P_M = 0.55$		
	Global	Shared	Constant	Global	Shared	Constant
E-M	227.1	233.9	230.5	220.0	228.0	221.5
Backdoor	131.3	140.4	122.9	127.6	136.8	119.2
Spyware	111.9	122.6	87.9	112.5	121.9	91.7
Dotstar.05	144.6	162.6	132.1	145.0	162.2	139.5
Dotstar.1	111.0	122.4	86.8	111.9	122.6	95.2
Dotstar.2	53.1	59.2	25.0	12.8	13.2	10.6
Dataset	$P_M = 0.75$			$P_M = 0.95$		
	Global	Shared	Constant	Global	Shared	Constant
E-M	209.0	216.2	212.5	169.6	174.8	171.4
Backdoor	113.6	122.4	110.3	94.5	101.3	91.2
Spyware	111.4	120.5	89.5	96.6	101.8	77.3
Dotstar.05	108.1	116.3	105.9	113.6	119.4	111.0
Dotstar.1	111.6	119.8	90.0	91.8	96.2	76.6
Dotstar.2	53.0	58.8	26.8	12.7	13.2	10.6

due to the serialization of the constant memory access. Since each thread in a block reads data from different tables (and different memory addresses), all the memory accesses in a half thread warp will be serialized. In the case of *E-M,* only one thread per block will be active and so constant memory accesses won't be serialized.

3.6.3 Multiple Tables vs. Single Table

In Table 3.5 we show the results of performing regular expression matching on GPU using a single alphabet transition table. The following observations can be made. First, the implementation of single table on global memory has throughput improvement over the multiple alphabet transition tables one. Since a single alphabet transition table only occupies 256 bytes of memory (one byte for each character in ASCII table), the range of memory accessed for alphabet translation is limited to 256 bytes. Therefore, single table provides much larger potential of the GPU global memory coalescing which contributes

Table 3.5: Throughput (in Mbps) obtained with single table and different memory implementation.

Dataset	$P_M = 0.35$			$P_M = 0.55$		
	Global	Shared	Constant	Global	Shared	Constant
E-M	230.5	237.4	235.6	223.9	230.5	228.0
Backdoor	147.3	150.1	153.1	142.0	145.0	146.0
Spyware	133.3	141.4	141.4	133.3	141.1	140.4
Dotstar.05	174.3	179.2	178.7	171.4	177.7	176.7
Dotstar.1	133.6	141.4	141.4	136.8	140.4	141.1
Dotstar.2	65.1	71.0	70.9	13.4	13.8	13.8
Dataset	$P_M = 0.75$			$P_M = 0.95$		
	Global	Shared	Constant	Global	Shared	Constant
E-M	215.5	216.9	219.2	174.8	173.8	175.7
Backdoor	126.1	128.7	131.1	98.9	103.8	103.5
Spyware	133.9	140.7	140.7	111.6	115.0	115.4
Dotstar.05	120.5	124.1	124.1	125.6	127.6	128.1
Dotstar.1	134.1	140.1	138.6	104.2	109.2	108.3
Dotstar.2	64.8	71.9	72.1	13.4	13.8	13.8

to performance gain. Higher cache hit rate achieved by single table also has positive effect on the results. Second, we can see the performance improvements by using shared memory. Because the GPU kernel needs to copy table information from global memory to shared memory at the beginning for each packet, better performances can be explained by fewer write operations on shared memory during each iteration. Third, the constant memory implementation benefits largely from single table and its performances are quite close to those of the shared memory implementation. In our single table implementation, the threads in a half warp are much more likely to access the same constant memory address. So there will be much less memory access serialization and the broadcast mechanism can further save certain amount of memory traffic. However, the weakness of single table is obvious compared to multiple-table implementation. We can see from Table 3.3 that merging alphabet transition tables will always create a single table with larger number of character classes, thus requiring more global memory to store DFA state transition table.

3.6.4 Effect of Number of DFAs

Table 3.6: Characteristics of DFAs generated by double set implementation

Dataset	# DFA	# Total states	# Classes (before merging)	# Classes (after merging)
E-M	1	28744	88	88
Backdoor	11	1709391	24 ~ 73	111
Spyware	16	1374986	32 ~ 61	89
Dotstar.05	11	1064994	74 ~ 87	90
Dotstar.1	21	1004455	71 ~ 81	90
Dotstar.2	38	1367862	60 ~ 77	90

In this section, we analyze how the number of DFAs affects the overall performance. In Table 3.6, we can see the DFAs generated by our double set implementation. Table 3.7 shows the corresponding performance results and compares them with the original uncompressed version. Similar to the results in Table 3.5, shared memory and constant memory achieve competitive performances while the global memory-based implementation is not as good as the other two. We can also notice that in some cases, the implementation with alphabet reduction leads to higher throughput than the uncompressed one. This is because the smaller DFA state transition table can lead to more regular memory access patterns and therefore higher cache hit rate. By comparing Table 3.5 and Table 3.7, we can see that fewer DFAs result in better performances for the same pattern-set. As mentioned before, the number of threads per block to conduct pattern matching is reduced with smaller number of DFAs. In the GPU kernel function, each thread needs to check whether an accepting state is reached, which potentially leads to branch divergence. Also, each thread performs an atomic operation to record the match information further impacts the matching speed. Therefore, larger number of threads causes poorer performance. In general, from the experiments, we can conclude that the best performances are achieved when minimum number of DFAs that fit the available

Table 3.7: Throughput (in Mbps) obtained from DFAs generated by double sets implementation.

Dataset	Uncompressed	Global	Shared	Constant	Uncompressed	Global	Shared	Constant
	P_M=0.35				P_M=0.65			
E-M	236.5	230.5	237.4	235.6	229.6	223.9	230.5	228.0
Backdoor	-	154.6	159.7	157.7	-	150.9	153.8	152.7
Spyware	178.2	177.7	185.6	185.6	179.2	180.3	182.4	185.0
Dotstar.05	186.1	186.7	191.2	193.6	187.2	182.9	187.8	190.7
Dotstar.1	158.9	150.9	160.1	158.9	163.8	156.9	165.6	164.3
Dotstar.2	71.3	65.1	70.4	70.8	14.1	13.5	13.8	13.8
	P_M=0.75				P_M=0.95			
E-M	219.2	215.5	216.9	219.2	176.2	174.8	173.8	175.7
Backdoor	-	133.9	135.6	134.4	-	106.3	107.2	106.5
Spyware	172.4	168.2	176.2	175.2	138.3	132.5	136.8	137.7
Dotstar.05	130.8	128.1	129.5	131.9	133.3	131.3	131.9	133.3
Dotstar.1	161.2	153.1	158.5	160.1	119.8	116.3	118.3	118.9
Dotstar.2	71.4	63.9	70.9	71.8	13.7	13.5	13.5	13.6

memory capacity is used. Another aspect to point out is that the total size of the DFAs generated for *backdoor* pattern-set before alphabet reduction is about 1.67GB, which is larger than our GPU global memory size. Therefore, the uncompressed approach is unable to process this pattern-set. Alphabet reduction can make this possible by reducing the size by nearly half.

3.6.5 Comparison of DFAs generation algorithms

In this section, we compare different DFAs generation algorithms discussed in Section 3.5. In Figure 3.2, we can see that the results of single set implementation are not as good as the others when dealing with complex pattern-sets like *spyware*. In *spyware* pattern-set, about 6.5% of the rules have weight larger than 3 and combining them with large amount of simple rules can cause large DFA state replication. Therefore, the number of DFAs generated for *spyware* using single-set implementation is relatively large. The number of DFAs generated using the single set implementation for *backdoor* and *dotstar0.2* pattern-sets are also affected by the combination of complex regular expressions and large amount of simple regular expressions. By comparing these two algorithms, we can conclude that the double set algorithm is general enough to deal with both simple and complex pattern-sets. On the other hand, we need to identify the threshold to be used to divide *simple* and *complex* sets. According to the weight distributions of regular expressions in different pattern-sets shown in Table 3.2, we conduct experiments to see how the performances are affected by the weight threshold that distinguishes simple regular expression from complex one. As can be seen in Table 3.2, if we define the threshold to be 2, the percentage of complex regular expressions is not larger than 9% for all the pattern-sets. By separating the small amount of complex regular expressions from large number of simple regular expressions, we can avoid too many state replications. When the threshold becomes larger, more complex regular expressions are combined with simple regular expressions. Therefore, the number of DFAs generated tends to be larger.

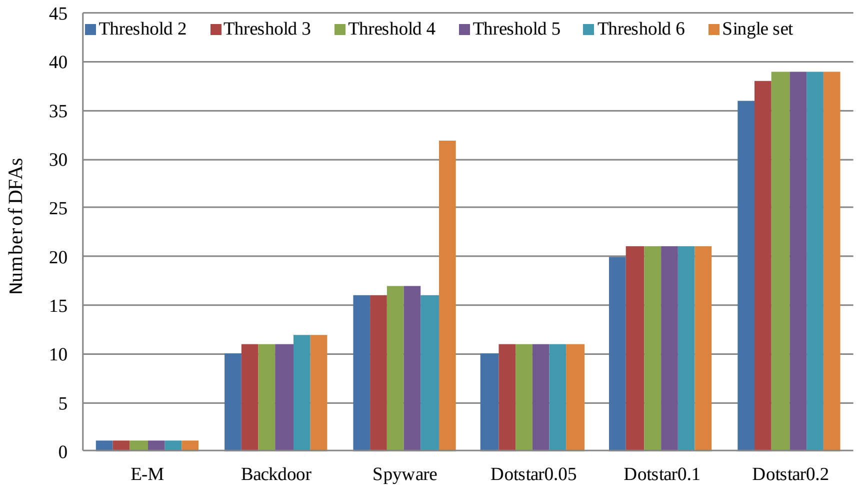


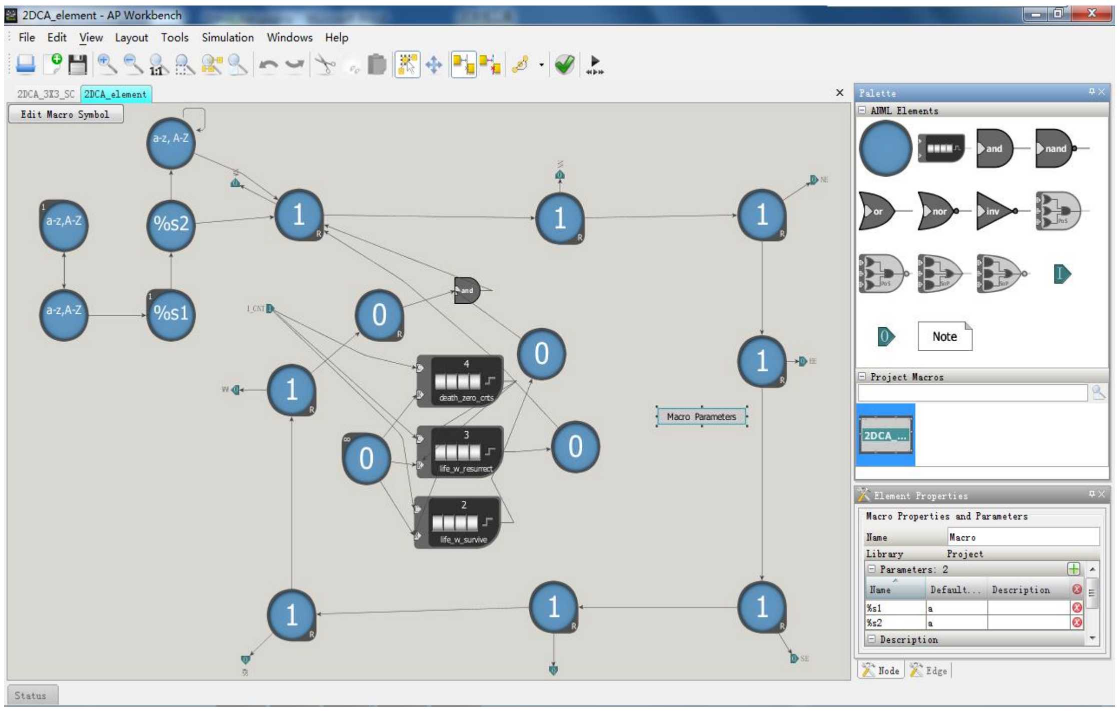
Fig. 3.2: Comparison of number of DFAs generated by two algorithms.

Chapter 4 ANML Implementation

One of the central tasks in our research is related to pattern matching on Micron's Automata Processor. Recall that this platform can be programmed by using ANML, an XML-based language to construct Automata Networks. Micron Technology also developed a graphic tool called ANML workbench to construct Automata Networks and export Automata Networks to XML files. We give an introduction to ANML workbench in Section 4.1. We develop tools to parse XML files and construct Automata Networks by using our C++ data structures. We show these data structures in Section 4.2. We created an ANML parser that allows transforming Automata Networks represented by these data structures for further optimizations (such as NFA reduction and alphabet reduction). We describe ANML parser in Section 4.3. In Section 4.4, we discuss a programming interface called ANML generator to generate Automata Networks using our data structures. Finally, in Section 4.5, we discuss optimization techniques for Automata Networks.

4.1 ANML Workbench

As shown in Figure 4.1, ANML workbench is the graphic tool to construct Automata Networks. We can use the available elements including STEs, counters and boolean elements in the upper right window on ANML workbench. We can also configure the properties for these elements, such as start properties of STEs, symbol sets of STEs, target count of counters and counter types of counters, in the lower right



Macro Properties and Parameters

Name	Macro
Library	Project

Parameters: 2

Name	Default...	Description
%s1	a	
%s2	a	

Description

Fig. 4.1: ANML workbench.

window. We can add transitions between elements by drawing lines connecting them. Furthermore, we can encapsulate a particular pattern in a block called Macro object and then replicate Macro objects to represent the patterns that appear frequently in our designs. Therefore, the Macro object increases the reusability of patterns on ANML workbench and makes our design more convenient. The block in the middle right window is a Macro object that stores the pattern shown in the left window. To check the correctness of our designs, ANML workbench provides a simulator to simulate the matching process for the user provided input stream. In addition, we can use ANML workbench to export our designs to XML files in the format shown in the left of Figure 4.7.

4.2 Data Structures for ANML Parser and Generator

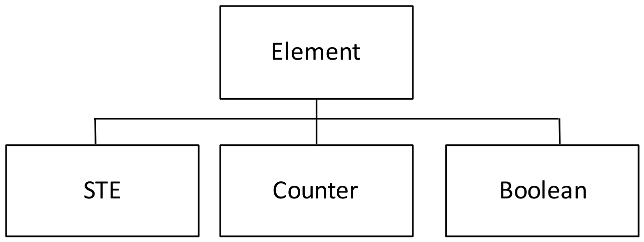


Fig. 4.2: Organization of data structures to represent Automata Networks.

We use ANML parser to parser the XML files generated by ANML workbench and represent corresponding Automata Networks using our C++ data structures. On the other hand, we use ANML generator as the programming interface to construct Automata Networks. Therefore, both ANML parser and generator rely on the data structures we used to represent Automata Networks. In Figure 4.2, we show the general organization of our data structures. We create a base class *Element* to achieve the features shared by all the three elements. We then use 3 child classes inheriting from *Element* to represent STEs, counters, and boolean elements.

Figure 4.3 shows the base structure of *Element* class. The general design of *Element* class has the variable *type* to indicate which type this element belongs to. It also stores transition information between different elements and has the variable *report* to indicate if it is an accepting element. The *ANFA* class in Figure 4.4 has variables that are unique to STEs. In particular, ANML parser stores symbol information associate to STEs in the variable s*ymbols*. The *start* variable shows whether the STE is an all-input, start-of-data or normal state. If the STE keeps being active after once being activated, the *latch* variable is set to be true. In Figure 4.5, the *counter* class stores the information about target count in the variable *count.* We use the variable *at_target* to determine the type of counter (roll, pulse and latch). Similar to *counter* class, we use *boolType* to store the type of boolean elements in the boolean class shown in Figure 4.6. In addition to variables shown in these classes, we also implement various functions related to the construction of Automata Networks based on these variables, such as defining the symbol set associated to STEs, adding transitions between different elements and so on.

```
Class Element {
      //type of element
       int type;

     //labeled transitions to STEs
      set<pair<symbol, Element*> > transition_pair;

     //transitions to count port of counters and boolean elements
      set<Element*> transition;

    //transitions to reset port of counters
     set<Element*>  reset_tx;

    //if it is an accepting element
     bool report;
}
```

Fig. 4.3: Structure of Element class.

```
Class ANFA:public Element
{
       // symbol set
       int_set *symbols;

       // start info
       string start;

       //latch info
       bool latch;
}
```

Fig. 4.4: Structure of STE class.

```
Class counter:public Element {
        /* target count number */
       int count;

       /* state upon activation */
      string at_target;
}
```

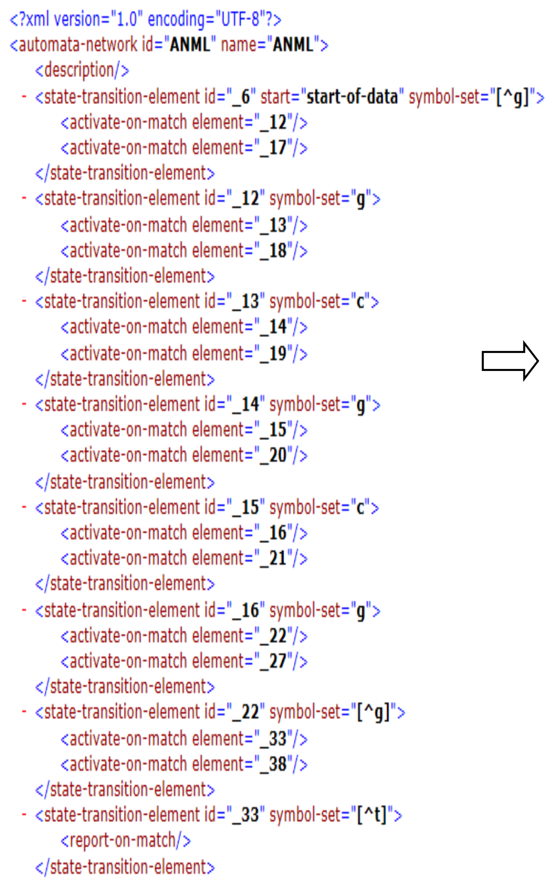
Fig. 4.5: Structure of counter class.

```
Class boolean:public Element {
        /* boolean type */
       string boolType;
}
```

Fig. 4.6: Structure of boolean class.

4.3 ANML Parser

The ANML parser allows transforming ANML XML files to our internal data structures shown in Section 4.2. Figure 4.7 shows a simple example of the mapping between the XML file and its corresponding Automata Network. Libxml2 is a XML C parser library that is developed for the Gnome project. Because of its easy usage and high portability, we use Libxml2 to extract the Automata Network information, including the characteristics of elements and edges, from XML files. To have the intact Automata Network structure stored in the XML file, the ANML parser takes all the characteristics of these elements described in Section 2.4.2 into account. ANML parser makes it convenient for us to apply further optimization algorithms that will be described in Section 4.4 to the patterns constructed on ANML workbench and enables the further conversion to Verilog files targeting FPGA implementation.



```
<?xml version="1.0" encoding="UTF-8"?>
<automata-network id="ANML" name="ANML">
   <description/>
 - <state-transition-element id="_6" start="start-of-data" symbol-set="[^g]">
      <activate-on-match element="_12"/>
      <activate-on-match element="_17"/>
   </state-transition-element>
 - <state-transition-element id="_12" symbol-set="g">
      <activate-on-match element="_13"/>
      <activate-on-match element="_18"/>
   </state-transition-element>
 - <state-transition-element id="_13" symbol-set="c">
      <activate-on-match element="_14"/>
      <activate-on-match element="_19"/>
   </state-transition-element>
 - <state-transition-element id="_14" symbol-set="g">
      <activate-on-match element="_15"/>
      <activate-on-match element="_20"/>
   </state-transition-element>
 - <state-transition-element id="_15" symbol-set="c">
      <activate-on-match element="_16"/>
      <activate-on-match element="_21"/>
   </state-transition-element>
 - <state-transition-element id="_16" symbol-set="g">
      <activate-on-match element="_22"/>
      <activate-on-match element="_27"/>
   </state-transition-element>
 - <state-transition-element id="_22" symbol-set="[^g]">
      <activate-on-match element="_33"/>
      <activate-on-match element="_38"/>
   </state-transition-element>
 - <state-transition-element id="_33" symbol-set="[^t]">
      <report-on-match/>
   </state-transition-element>
```

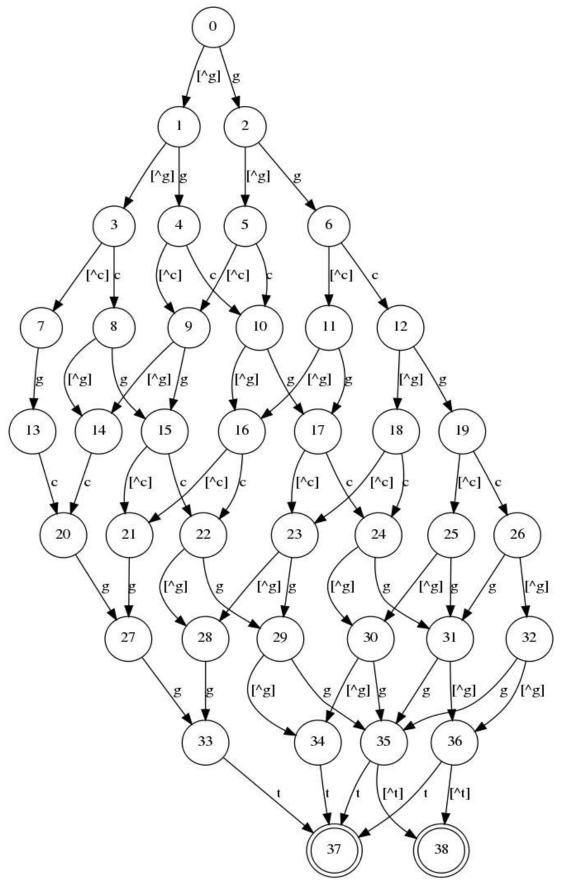


Fig. 4.7: Transfer from XML file to Automata Network.

4.3.1 API functions of ANML parser

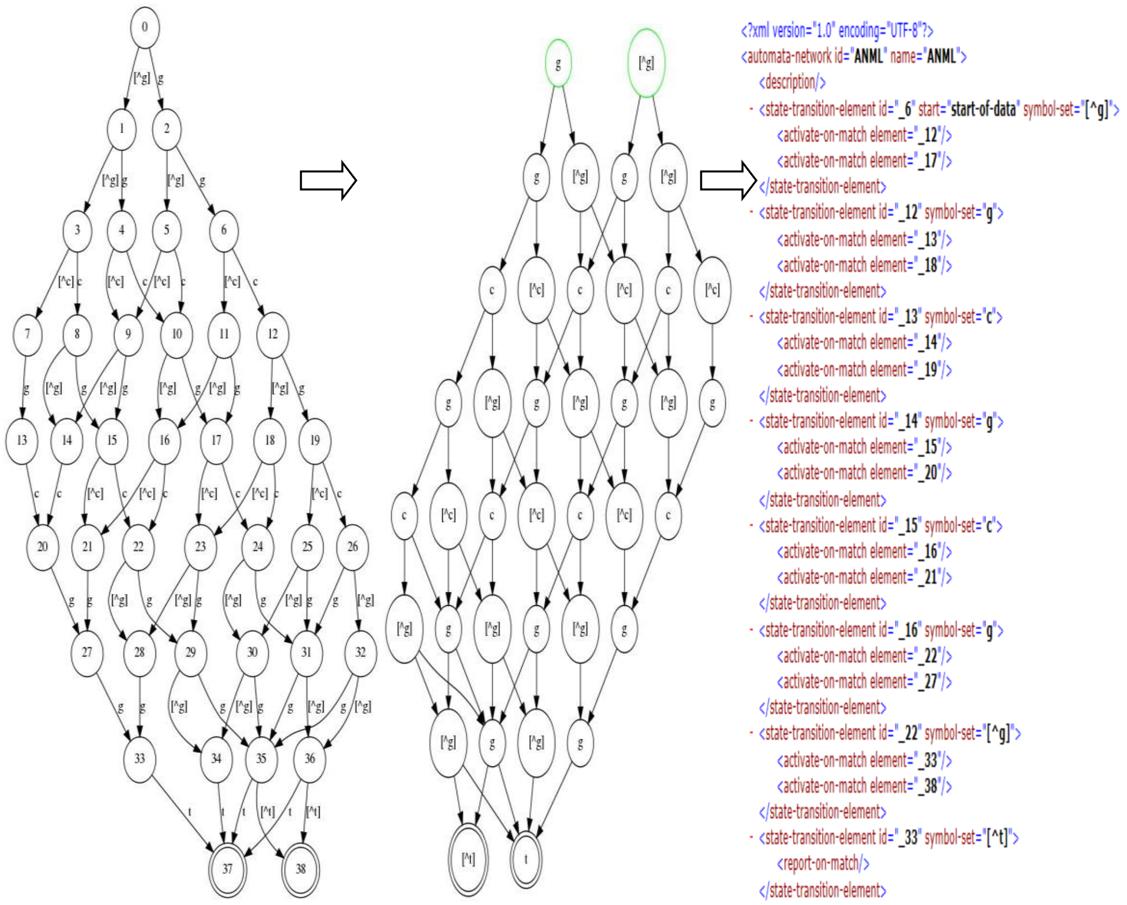
```
//parse example.xml to our data structures
anml_parser *ap = new anml_parser();
Element* elem = ap->parse("example.xml");
```

Fig. 4.8: API functions of ANML parser.

We show the API functions of ANML parser in Figure 4.8. The *anml_parser* class implements all the functions related to the parsing of XML files by using functions in Libxml2 library. We only need to provide the name of XML file(example.xml in this case) to the *parse* function and the start element *elem* of the Automata Network is returned by this function.

4.4 ANML Generator

Micron Technology developed ANML workbench to construct Automata Networks. However, when the design has large amount of elements and complex transitions between elements, we have to make large efforts to construct the design manually. Since the easy construction of Automata Networks is important for usability, we provide the programming interface called ANML generator to make it easier for users to construct their own Automata Networks. By simply utilizing the data structures we have, we can create element objects and add transition between them. In this way, we can create the corresponding pattern with small efforts. On the other hand, since we already have the NFA generation tool allowing different optimizations, such as alphabet reduction, NFA state reduction and stride doubling which will be described in Section 4.5. We can apply these optimizations on the Automata Networks and use ANML generator to export optimized Automata Networks to XML files. Therefore, we can further deploy Automata Networks stored in XML files on the Automata Processor. The detailed example is illustrated in Figure 4.9. It shows the conversion from the representation using our data structures to ANML representation. ANML generator takes the existing Automata Network and exports it to the XML file which is compatible with ANML workbench. In order to check the correctness of the pattern we generated, ANML generator provides a way to visualize the Automata Network structural information with diagrams of abstract graphs. Graphviz, an open source graph visualization library, is used to achieve this visualization purpose and export the structure to corresponding JPEG file.



```
<?xml version="1.0" encoding="UTF-8"?>
<automata-network id="ANML" name="ANML">
   <description/>
 - <state-transition-element id="_6" start="start-of-data" symbol-set="[^g]">
      <activate-on-match element="_12"/>
      <activate-on-match element="_17"/>
   </state-transition-element>
 - <state-transition-element id="_12" symbol-set="g">
      <activate-on-match element="_13"/>
      <activate-on-match element="_18"/>
   </state-transition-element>
 - <state-transition-element id="_13" symbol-set="c">
      <activate-on-match element="_14"/>
      <activate-on-match element="_19"/>
   </state-transition-element>
 - <state-transition-element id="_14" symbol-set="g">
      <activate-on-match element="_15"/>
      <activate-on-match element="_20"/>
   </state-transition-element>
 - <state-transition-element id="_15" symbol-set="c">
      <activate-on-match element="_16"/>
      <activate-on-match element="_21"/>
   </state-transition-element>
 - <state-transition-element id="_16" symbol-set="g">
      <activate-on-match element="_22"/>
      <activate-on-match element="_27"/>
   </state-transition-element>
 - <state-transition-element id="_22" symbol-set="[^g]">
      <activate-on-match element="_33"/>
      <activate-on-match element="_38"/>
   </state-transition-element>
 - <state-transition-element id="_33" symbol-set="[^t]">
      <report-on-match/>
   </state-transition-element>
```

Fig. 4.9: ANML generator workflow.

4.4.1 API functions of ANML generator

```
//create symbol set
int_set *chars = new int_set();
//add symbol 'a'
chars->insert(97);

ANFA *nfa1 = new ANFA();
ANFA *nfa2 = new ANFA();
//add transition between STEs
nfa1->add_transition(chars, nfa2);

Counter *cnt = new Counter();
//set target count
cnt->set_count(3);
//set counter type to roll counter
cnt->set_at_target("roll");
//add transition to the count port of counter
nfa1->add_transition(cnt);
//add transition to the reset port of counter
nfa2->add_transition(cnt, true);

Boolean *boolean = new Boolean();
//set boolean type to "or"
boolean->set_type(3);
//add transition to boolean
nfa1->add_transition(boolean);
…
```

Fig. 4.10: Sample code to construct Automata Networks.

Table 4.1: Counter Types.

Counter Type
roll
pulse
latch

Table 4.2: Mapping between boolean elements and type numbers.

Boolean Type	*Type Number*
Inverter	2
Or	3
And	4
Nand	5
Nor	6
Sum of Products	7
Product of Sums	8
Not Sum of Products	9
Not Product of Sums	10

Figure 4.10 shows the sample code to use API functions in ANML generator to construct Automata Networks. We can first define the symbol set *chars* that leads to the transition from STE *nfa1* to STE *nfa2.* We need to insert an integer number corresponding to the position of the symbol in ASCII table to *chars*. We can also create counters by configuring the target count and type. We can see the counter types in Table 4.1. Furthermore, we can add transitions to both the count port and reset port of counters as shown in Figure 4.10. Similar to counters, we need to set the boolean type by providing the type number to *set_type()* function each time when we create boolean elements. Table 4.2 shows the mapping between boolean elements and type numbers.

Besides the construction of Automata Networks, ANML generator provides the feature of converting Automata Networks to XML files. Therefore, XML files can be further compiled and deployed on the Automata Processor. In Figure 4.9, we can see that our data structures use labeled transition for STEs. However, Automata Networks created on ANML workbench associate symbols with STEs rather than with transitions and XML files store information of Automata Networks in the same format. In order to export our data structures to XML files, we thus first convert our data structures to the corresponding structures that integrate symbols with STEs and then we can export these converted structures to XML files directly. We use the sample code in Figure 4.11 to achieve the conversion from Automata Networks represented by our data structures to XML files.

```
//convert our structures of Automata Networks to intermediate structures
ANML_ELEMENT *anml_elem;
anml_elem=nfa1->to_ANML();

//export intermediate structures to XML file named "ANML"
anml_elem->to_xml(file,"ANML");
```

Fig. 4.11: Sample code to export Automata Networks to XML file.

4.5 Optimization Techniques

Since useful optimization techniques originally designed for classical NFA can also be useful to Automata Networks that has only STEs, we integrate efficient NFA optimization techniques in [6] to our programming interface.

The first optimization is called NFA reduction algorithm [6]. As can be observed in Figure 4.12, this algorithm causes common prefixes to merge and reduces the total number of states in the NFA. Therefore, NFA reduction algorithm leads to more compact NFA which will contribute to the implementations on both FPGA and the Automata Processor. We will discuss the FPGA implementation of regular expression matching and show how the size of NFA can affect the overall performance of FPGA in Chapter 5.

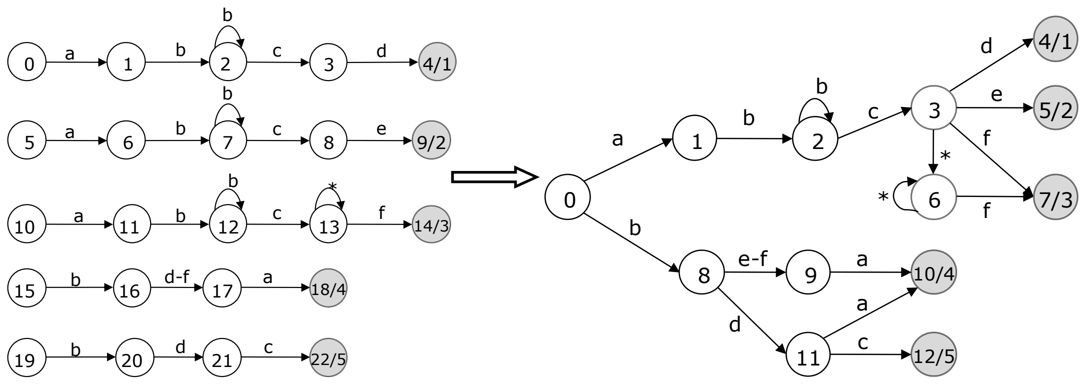
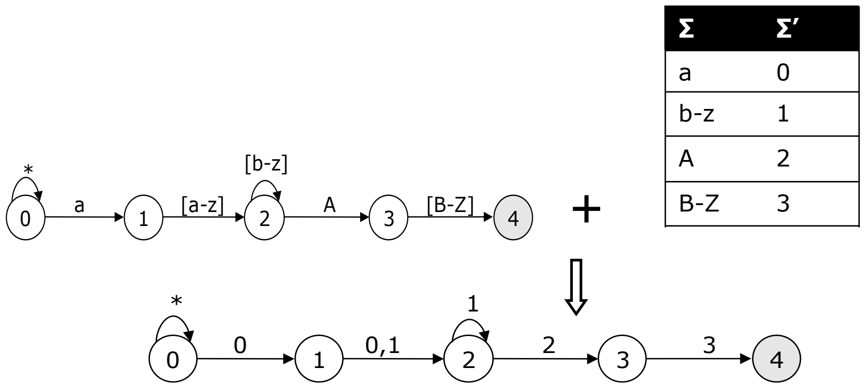


Fig. 4.12: Effect of NFA reduction algorithm. Accepting states are colored grey.

Secondly, we can apply alphabet reduction to reduce alphabet size as mentioned in Section 3.2. As shown in Figure 4.13, if we only use a small subset of symbols from ASCII table in the NFA, we can apply alphabet reduction algorithm to divide symbols into different classes. Therefore, the number of transitions per state is significantly reduced to be the number of classes. Alphabet reduction leads to NFA with smaller size and is important for implementations on both memory-based and logic-based hardware platforms.

Finally, we can reconstruct the original NFA to k-NFA [6] that receives k symbols at each transition by using the stride doubling algorithm [6]. Therefore, k symbols can be processed for each transition and thus the k-NFA is regarded as k-stride NFA. The k-NFA can achieve much higher throughput than the original NFA. However, given an NFA defined on alphabet $\sum$, the corresponding k-NFA is defined on alphabet $\sum^k$. In this case, alphabet reduction can be applied to reduce increased alphabet size efficiently.



Σ	Σ′
a	0
b-z	1
A	2
B-Z	3

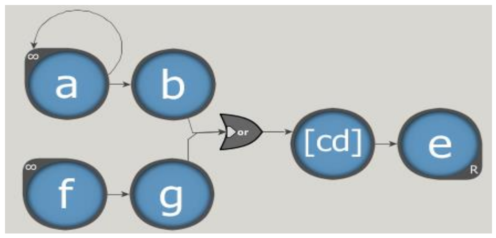
Fig. 4.13: Implementation of alphabet reduction algorithm. Accepting states are colored grey.

Chapter 5 FPGA implementation

In this chapter, we focus on the FPGA implementation of regular expression or Automata Network matching. In Section 5.1, we propose the general design for the FPGA implementation. In Section 5.2, we discuss optimization techniques. Finally, we conduct experiments on both synthetic and real datasets.

5.1 General Design

(a)



(b)

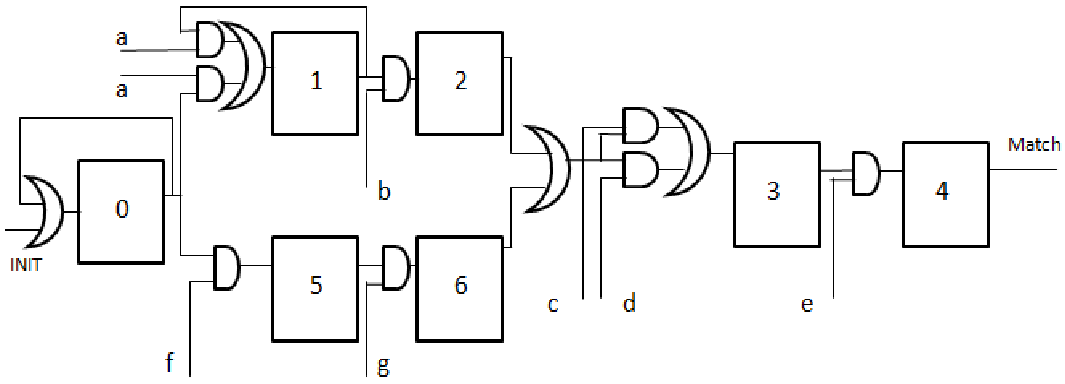


Fig 5.1: (a) Automata Network and (b) logic representation through one-hot encoding scheme. The INIT signal is asserted on the first character of a new input stream. The MATCH signal is asserted upon matching the pattern.

FPGA provides a promising platform for Automata Networks. To perform pattern matching on FPGA and compare the performances of Automata Processor- and FPGA-based implementations, we developed a Verilog generator that converts Automata Networks to Verilog files. These files can then be deployed on FPGA using traditional

HDL synthesis, map, place and route features provided by Xilinx ISE design suite [29].

FPGA implementation based on the one-hot encoding scheme [13] allows processing one input character per clock cycle independent of the number of active states. In one-hot encoding scheme, each NFA state is represented by a flip-flop and each symbol is represented by one bit. The output of the flip-flop representing a state is and-ed with the symbols on its outgoing transitions, and the resulting signals are routed toward the flip-flops representing the target states. This basic scheme, first proposed by Floyd and Ullman in [13], is later used by most NFA-based implementations on FPGA. Automata Networks [11] extend NFA with counter and boolean elements. As mentioned in Section 2.4.2, the counter element in Automata Networks has three distinct types, namely roll, pulse and latch counters. We thus need to create three Verilog counter modules to achieve the functionality these three types of counters. Since the primitive modules for logic gates are already provided in Verilog, we can easily make use of these modules to represent the boolean elements in Automata Networks. We show an Automata Network and its logic representation using one-hot encoding scheme in Figure 5.1.

Note that, for any given clock cycle, the NFA active set is represented by the set of flip-flops which are concurrently active. Moreover, interconnections allow multiple state transitions to occur in a single clock cycle.

The overall design schematic is represented in Figure 5.2. Besides the clock and an *INIT* signal, which is set at the beginning of every input stream, the module receives *k* characters at every clock cycle. The output of the module is a set of signals representing the output of accepting elements and therefore the match of the corresponding regular expressions. The first block stores the alphabet reduction information which will be

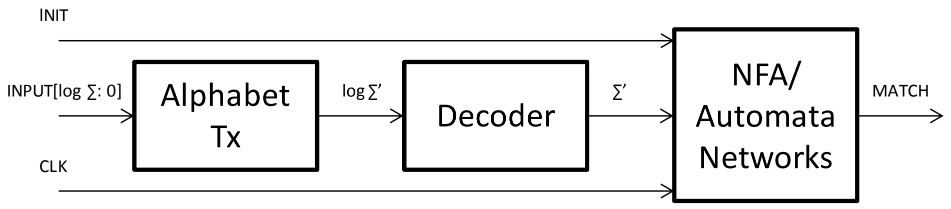
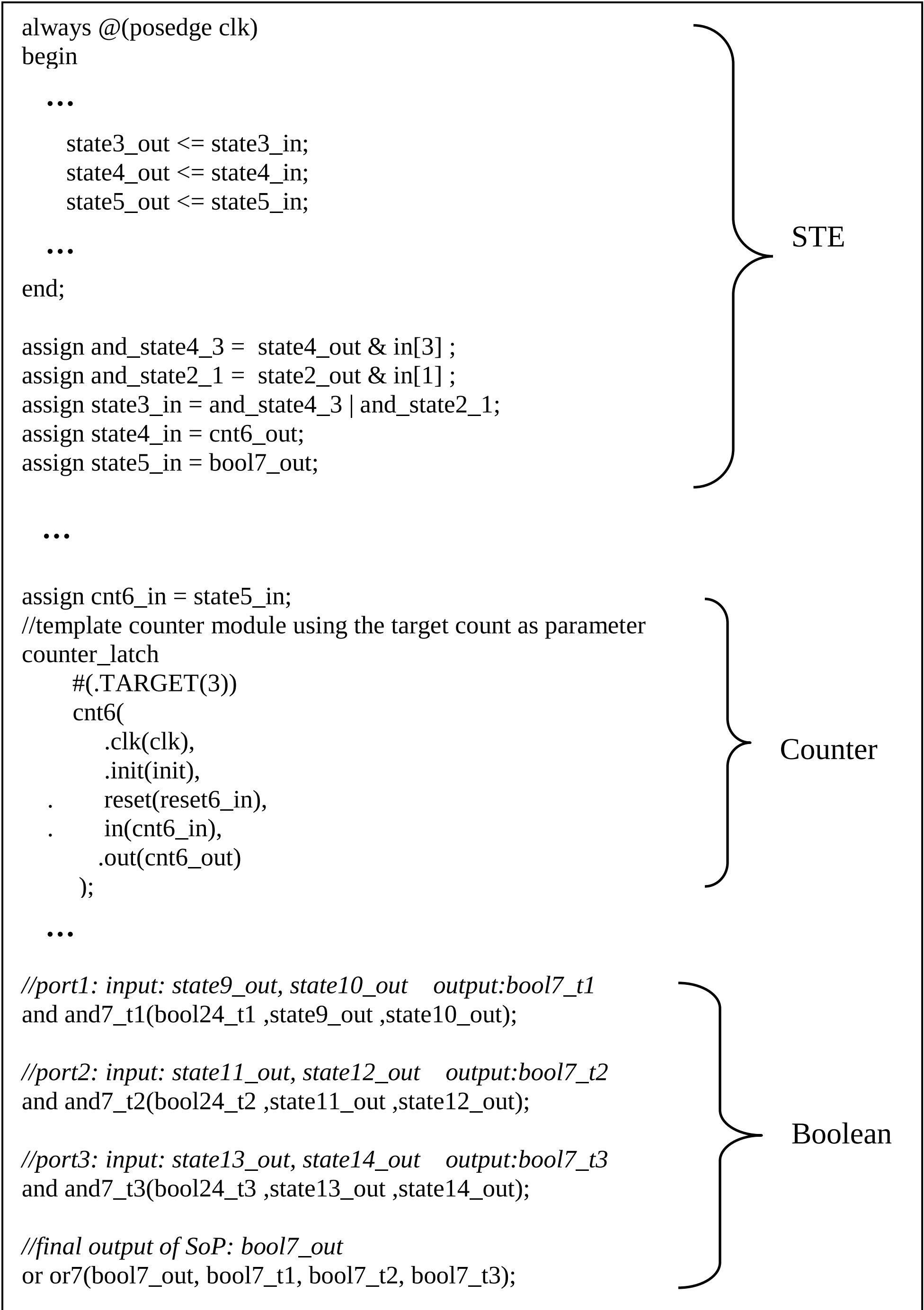


Fig 5.2: General FPGA design.

described in Section 5.2. The output of this block must be decoded to produce a one-hot encoding of the processed character, which is the input to the NFA block. This operation is performed by the alphabet decoder. The NFA/Automata Networks module is implemented as described in Figure 5.1.

We show the sample Verilog code that achieves the features of STEs, counters and boolean elements in Figure 5.3. First, we can assign the output of elements, including STEs, counters and boolean elements, to the input of next target elements. To represent labeled transitions, we use the *'&'* symbol in Verilog as the *and* gate for the state output and input symbol. In addition, the output of each state will be updated at every rising edge of *clock* signal. We also create 3 template counter modules, namely counter_latch, counter_pulse, and counter_roll, to represent three counter types. Therefore, we can reuse these template modules by providing different target numbers of count, input and output signals each time. Furthermore, as shown in Figure 5.3, we use the primitive logic modules provided in Verilog to achieve the logic functions of different boolean elements. For example, we use three *and* modules and an *or* module to represent the SoP boolean element in the code sample.



```
always @(posedge clk)
begin
   …
      state3_out <= state3_in;
      state4_out <= state4_in;
      state5_out <= state5_in;
   …
end;

assign and_state4_3 =  state4_out & in[3] ;
assign and_state2_1 =  state2_out & in[1] ;
assign state3_in = and_state4_3 | and_state2_1;
assign state4_in = cnt6_out;
assign state5_in = bool7_out;

   …

assign cnt6_in = state5_in;
//template counter module using the target count as parameter
counter_latch
       #(.TARGET(3))
       cnt6(
           .clk(clk),
           .init(init),
   .       reset(reset6_in),
   .       in(cnt6_in),
          .out(cnt6_out)
        );
   …

//port1: input: state9_out, state10_out    output:bool7_t1
and and7_t1(bool24_t1 ,state9_out ,state10_out);

//port2: input: state11_out, state12_out    output:bool7_t2
and and7_t2(bool24_t2 ,state11_out ,state12_out);

//port3: input: state13_out, state14_out    output:bool7_t3
and and7_t3(bool24_t3 ,state13_out ,state14_out);

//final output of SoP: bool7_out
or or7(bool7_out, bool7_t1, bool7_t2, bool7_t3);
```

Fig. 5.3: Verilog sample code to implement Automata Networks.

5.2 Design Optimizations

As discussed in [6], several optimizations can be performed to reduce resource utilization and lead to a better design.

First, we can apply *single input optimization* [6]. If one state only accepts a single transition, we can eliminate the use of logic gate and have the negation of input symbol directly connected to reset signal of flip-flop. Second, we use *multiple outputs optimization* [6] to further reduce logic utilization. If a state accepts a transition with a symbol set whose size is larger than a threshold from another state, the transition can be represented as a negation of symbol set. Both of these optimizations reduce the number of LUTs needed and facilitate wiring and routing.

Alphabet reduction [6] is used to reduce alphabet size and thus the number of LUTs used. In order to implement alphabet reduction on FPGA, we need to have alphabet translator to translate symbols into alphabet classes. The most straightforward and efficient approach to implement alphabet translator is using combinational logics on FPGA.

For classical NFA without counter and boolean elements, several additional optimizations can be applied. First, the NFA reduction algorithm [6] mentioned in Section 4.3 can be used to limit the number of states and thus the number of flip-flops used on FPGA. Second, hardware stride doubling [6] aims to improve regular expression matching throughput on FPGAs by having multi-level logic gates. In order to traverse multiple symbols at each clock cycle, we make use of multiple levels of logic gate while keeping the number of flip-flops at the same time. Finally, the algorithmic stride doubling [6] approach described in Section 4.3 can be used to reconstruct the original NFA to receive multiple symbols at each transition. Therefore, multiple symbols can be processed during each clock cycle on FPGA and the performance is improved significantly.

The above design optimizations are all integrated with our FPGA implementation to improve the overall performance.

5.3 Experimental Evaluation

In this section, we present our experimental evaluation of FPGA-based implementation of Automata Networks.

First, we create a synthetic NFA generator that can automatically generate NFA according to user-defined parameters (including the number of states and the average state outdegree). By configuring these parameters, we study how they can affect the overall performance of FPGA designs.

Second, we evaluate the FPGA implementation on two real pattern-sets, called *spyware* and *backdoor* which are used in Section 3.6 for GPU experiments.

Third, we evaluate our FPGA design using different Automata Networks, namely 1D and 2D cellular automaton. These datasets have been designed by Dr. Skadron's group at University of Virginia.

Finally, we estimate the conservative resource utilizations of different datasets on the Automata Processor according to the hardware resource descriptions of Automata Processor shown in Table 2.2. Since the hardware of Automata Processor is not available for experimentation, future work should focus on the comparisons of both resource utilization and performance between FPGA- and Automata Processor-based implementations.

To evaluate our design, we synthesized all these datasets on Xilinx Virtex-2P XC2VP20 [30], Virtex-4 XC4VLX25 [31] and Virtex-5 XC5VLX50 [32] devices. We used Xilinx ISE design suite, v. 10.1 [29].

5.3.1 Effect of Average Node Outdegree

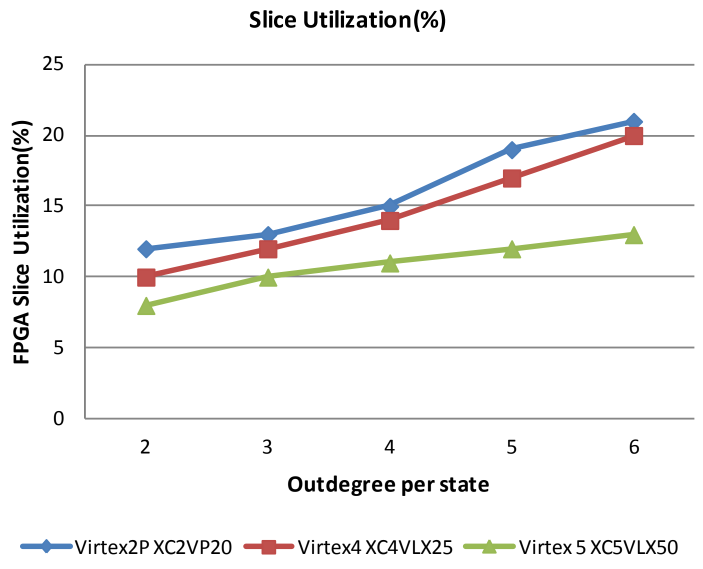


Fig. 5.4: Slice utilization of different average outdegrees per state.

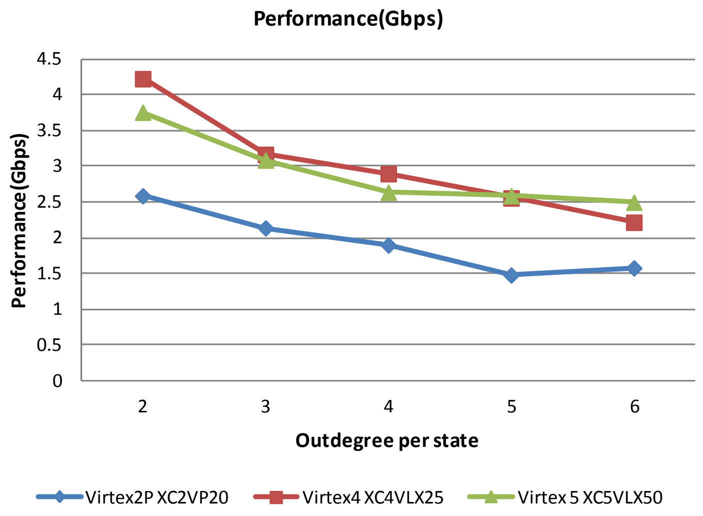


Fig. 5.5: FPGA performance of different average outdegrees per state.

Figure 5.4 and 5.5 shows the slice utilization and performance of NFA, consisting of 1000 states with different average node outdegrees. As we can see, while the number of average outgoing transitions increases from 2 to 6, the slice utilization increases and the performance decreases. Recall that transitions are implemented by LUTs in our FPGA implementation. Therefore, a larger number of average outgoing transitions per state translates to larger LUT utilization. Since a lower LUT utilization and amount of wires make the place and route operation more efficient and lead to a higher operating frequency, performance goes down when the avarage number of outgoing transitions per state increases. In addition, the estimated Automata Processor block utilization is about 4.16% which is smaller than the FPGA slice utilization.

5.3.2 Effect of Number of States

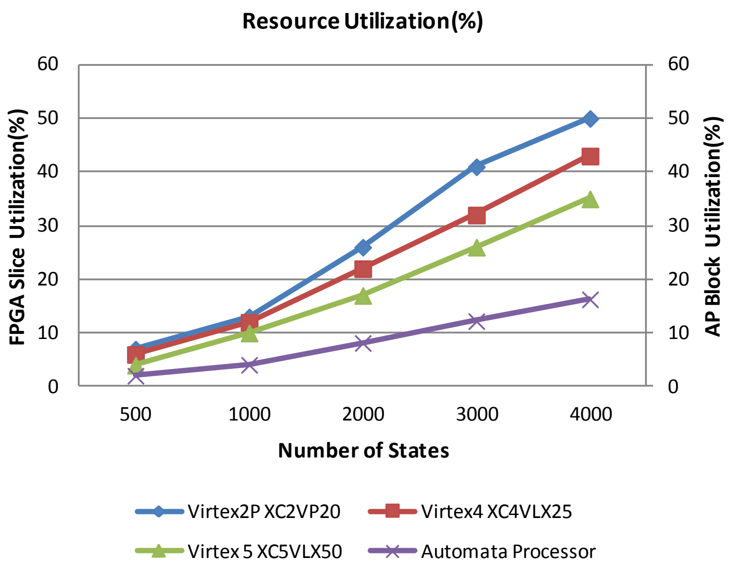


Fig. 5.6: Resource utilization of different number of states.

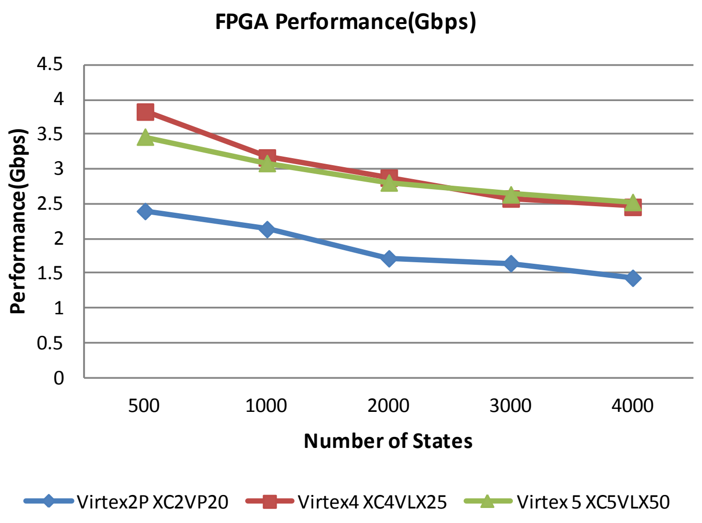


Fig. 5.7: FPGA performance of different number of states.

Figure 5.6 and Figure 5.7 show the effect of increasing the number of states when keeping the average node outdegree fixed. As the total numbers of states and transitions increase, the corresponding number of flip-flops and LUTs used also becomes larger. Similar to Figure 5.5, the higher performance is achieved from lower penalty on the operating frequency and larger flip-flop and LUT utilization will suffer from the overhead caused by more difficult wiring and routing. Therefore, FPGA implementation of pattern matching should focus on reducing the logic resources utilization and facilitating wiring and routing. Furthermore, as shown in Figure 5.6, the resource utilization of the Automata Processor is much smaller than that of FPGA for the same NFA.

5.3.4 Real Regular Expression Pattern-set Evaluation

In this section, we aim to evaluate the performance of regular expression matching on FPGA. Since the the NFA representations of the synthetic pattern-sets from Section

3.6.1 have large number of states, the amount of flip-flops needed exceeds the capacity of FPGA. Therefore, we select two real pattern-sets named *spyware* and *backdoor* for our expriments. As shown in Figure 5.8, *spyware* has the slice utilization around 60% on Virtex-5 XC5CLX50 device. In Figure 5.9 *spyware* achieves performance of 3.5 Gbps. On the other hand, although the slice utilization of *backdoor* is smaller than that of *spyware*, its performance is not as good as that of *spyware*. From the expriments, we find that the ratio between the number of LUTs and flip-flops used by *backdoor* is nearly 3 times larger than that of *spyware*. Therefore, the more difficult wiring and routing of *backdoor* result in lower clock frequency and thus worse performance. From our estimation, the block utilizations of *spyware* and *backdoor* on the Automata Processor are 32% and 20% respectivly. By comparing them with the data in Figure 5.8, we can see that both of these pattern-sets consume smaller amount of resources on the Automata Processor.

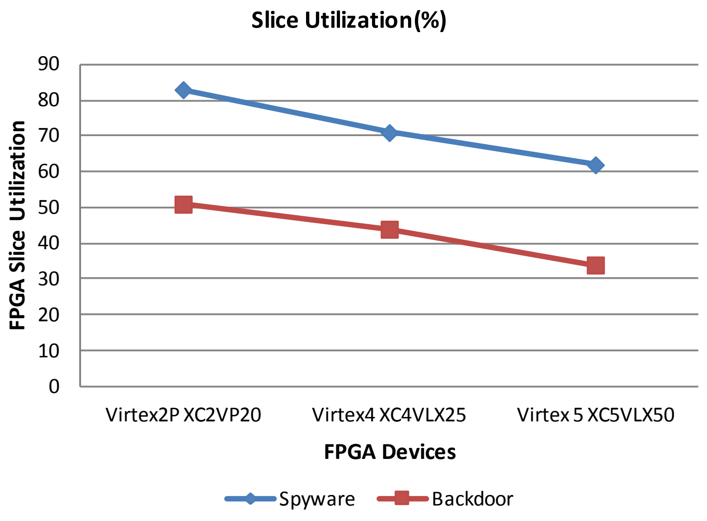


Fig. 5.8: Slice utilization of spyware and backdoor.

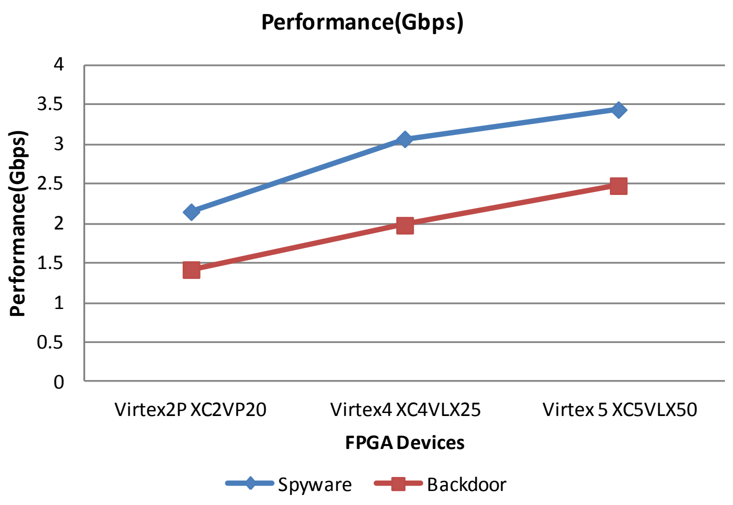


Fig. 5.9: FPGA performance of spyware and backdoor.

5.3.5 1D Cellular Automaton

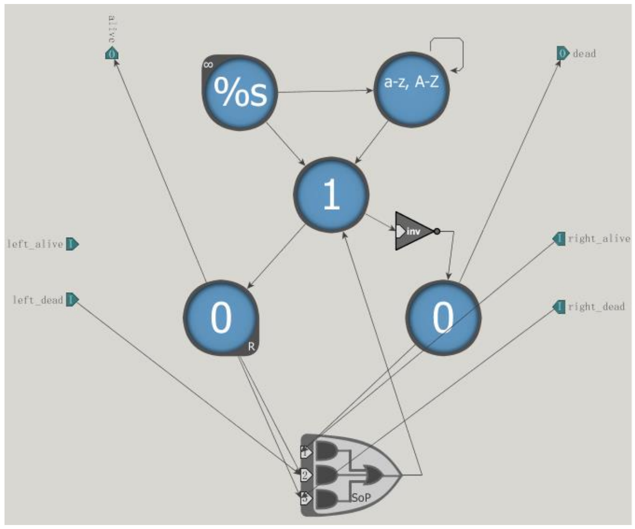


Fig. 5.10: Design of 1DCA cell.

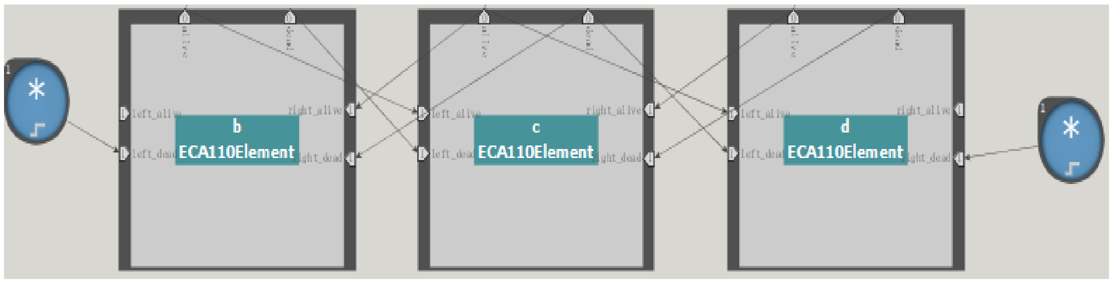


Fig. 5.11: 1DCA with 3 cells.

Table 5.1: Rule 110 for 1D Cellular Automaton.

current pattern	111	110	101	100	011	010	001	000
new state for center cell	0	1	1	0	1	1	1	0

The 1D cellular automaton (1DCA) is a one-dimensional cellular automaton where there are two possible states and a rule (rule 110 [28]) shown in Table 5.1 to determine the state of a cell in the next generation. Furthermore, the elementary cellular automaton with rule 110 is known to be Turing complete and capable of universal computation.

In general, as provided in Figure 5.10 and Figure 5.11, each cell can have the state of either alive or dead. The next state of each cell depends on the state of central element and the states of two neighbors. As we can see, the inputs from two neighboring cells and the cell itself to the SoP (Sum of Products) boolean element determine the next state of the current cell. Therefore, we can connect different number of cells to simulate the rule 110.

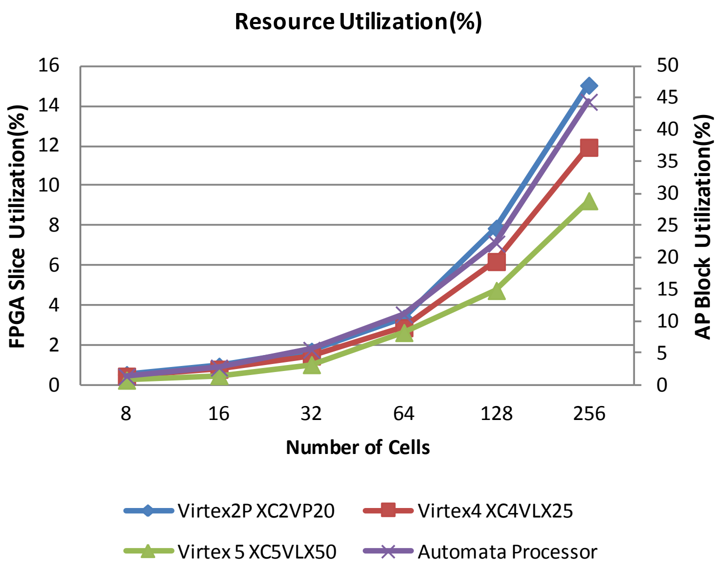


Fig. 5.12: Resource utilization of 1DCA.

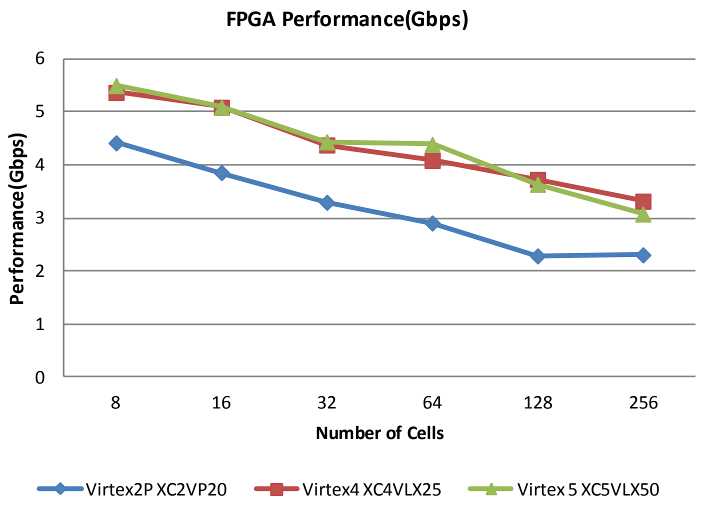


Fig. 5.13: FPGA performance of 1DCA.

We conduct experiments on FPGA using 1DCA consisting of 8~256 cells. As shown in Figure 5.12, the biggest 1DCA design consists of 256 cells and the slice utilization is around 15% on Virtex5 XC5VLX50. The resource utilization of the Automata Processor is almost 3 times larger than that of FPGA for the same size of design. At the same time, in Figure 5.13, the throughput that can be achieved for 256 cells is larger than 3 Gbps on both Virtex 4 XC4VLX25 and Virtex 5 XC5VLX50 devices.

5.3.6 2D Cellular Automaton

Table 5.2: Rule of 2D Cellular Automaton.

# of alive neighbors / Self status	<2	2	3	>3
Alive	Dead	Alive	Alive	Dead
Dead	Dead	Dead	Alive	Dead

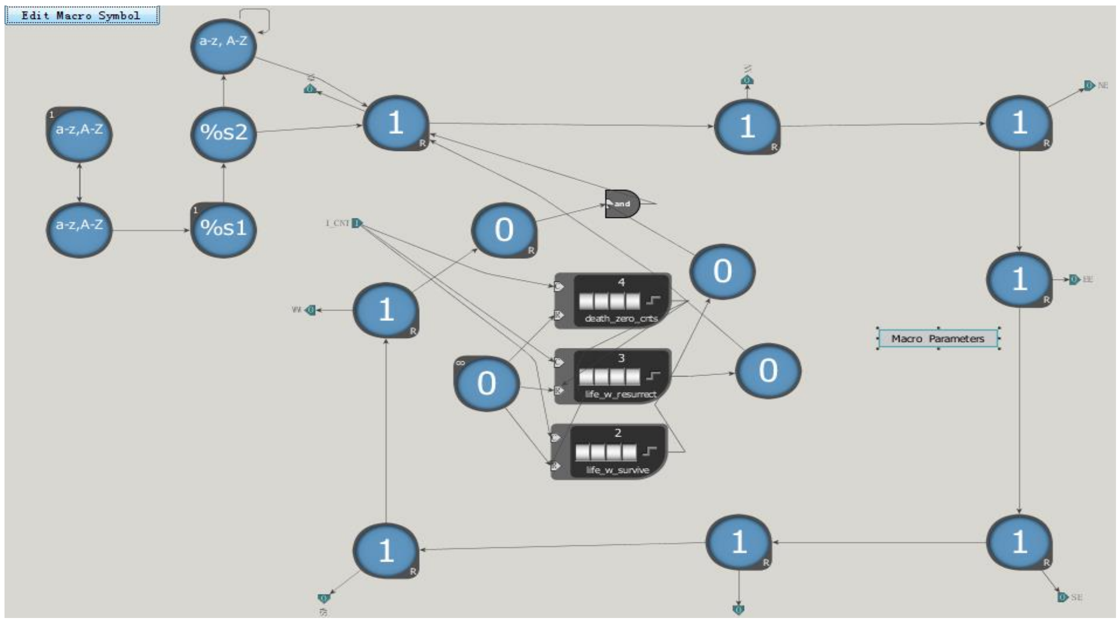


Fig. 5.14: Design of 2DCA.

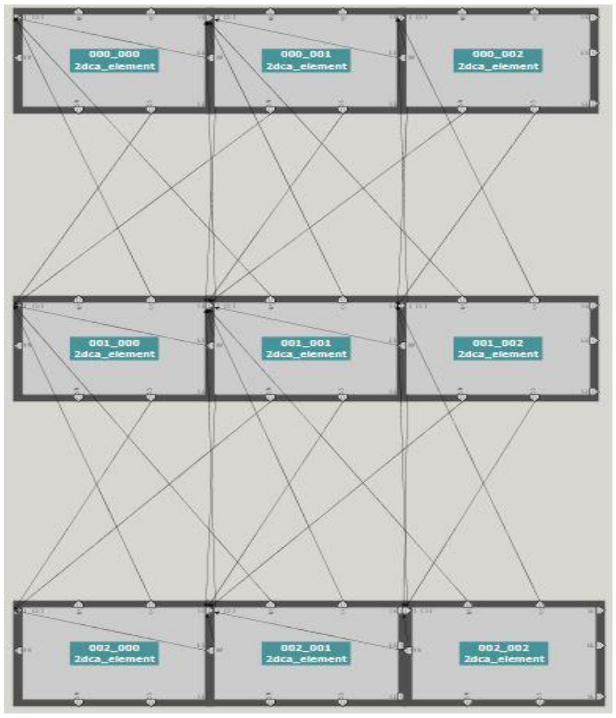


Fig. 5.15: 3x3 2DCA.

In this section, we evaluate the performance of 2D cellular automaton which implements the famous Game of Life [2]. Game of Life is an infinite two-dimensional orthogonal grid of square cells, each with the state of either alive or dead. Every cell interacts with its eight neighbors, which are the cells that are horizontally, vertically, or diagonally adjacent. At each step in time, transitions obeying the rule shown in Table 5.2 occur. As we can see from the design, counters are used to count the number of alive neighbors and determine the state of next generation. Game of Life is also known to be Turing complete. The general ANML design of Game of Life is shown in Figure 5.14 and Figure 5.15.

The resource utilizations of the 2DCA implementation on FPGA and Automata Processor are shown in Figure 5.16. We show the FPGA performance in Figure 5.17. 2DCA with 8x12 cells is the largest feasible size that can be implemented on the Automata Processor. As we can see from the figure, its slice utilization is around 20~30% for different FPGA devices. When the size of design is larger than 15x16, the slice utilization of Virtex2 and Virtex4 becomes 99% and 88% respectively. In this case, we can also see the significant performance loss on all three devices. In addition, each cell has 3 counters in the design and their target counts are 2, 3 and 4. The larger target count a counter has, the larger number of flip-flops is needed to represent it. Therefore, the relatively low slice utilization is due to these small target counts. However, automaton designs using counters with larger target counts would lead to higher slice utilization.

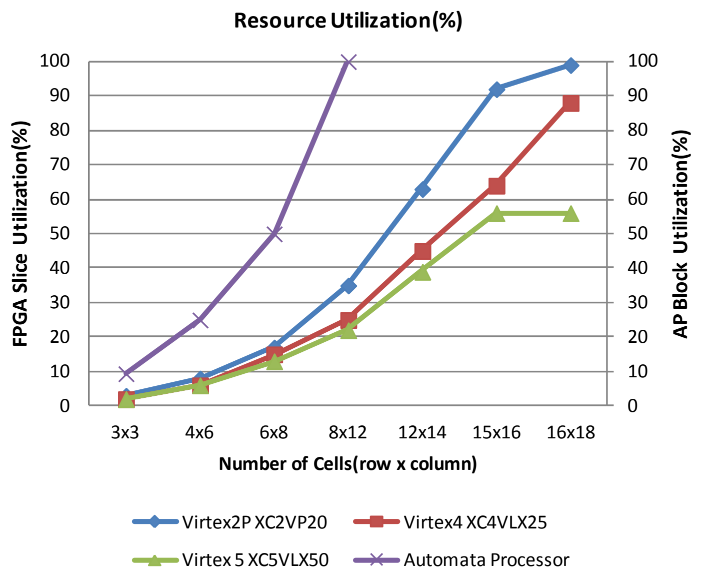


Fig. 5.16: Resource utilization of 2DCA.

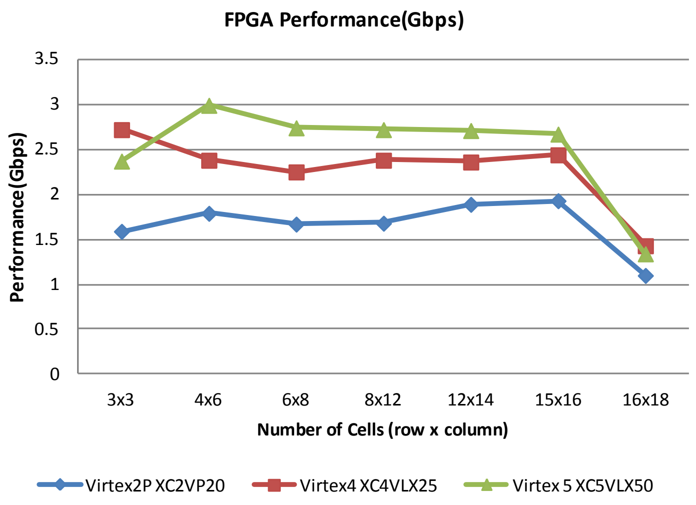


Fig. 5.17: FPGA performance of 2DCA.

Chapter 6 Summary

Regular expression matching is a central task in several networking applications, where the packet payload must be inspected against sets of patterns of interest. Hardware implementations of regular expression matching engines fall into two categories: memory-based and logic-based solutions. In both cases, the design aims to maximize the processing throughput and minimize the resources requirements, either in terms of memory or of logic cells. In this thesis, we study both the memory-based and logic-based solutions.

First, as the memory-based implementation of regular expression matching, GPU implementation achieves good performance because of its massive parallelism. In this work, we consider different approaches to regular expression matching on GPUs. To this end, we have used datasets of practical size and complexity and explored advantages and limitations of DFA-based implementations. Our evaluation shows that, because of the regularity of its computation, an uncompressed DFA solution achieves good performance and is scalable in terms of the number of packet-flows that are processed in parallel. However, on large and complex datasets, such representation may lead to exceeding the memory capacity of the GPU. By dividing regular expressions into multiple clusters and applying alphabet reduction to DFAs, we can alleviate the memory pressure efficiently without much performance impacts.

Second, we present our FPGA implementation. We convert both classical NFA and Automata Network to the logic representations on FPGA and conduct pattern matching on real and synthetic datasets. Furthermore, according to the experiment results, FPGA proves to be a preferred platform to conduct regular expression matching. However, the bottom neck of FPGA implementation comes from the limited number of flip-flops. When the number of NFA states becomes large, it may exceed the FPGA capacity. In addition, low reconfigurability, higher power consumption and a lack of scalability in the number of concurrent flows add constraints to FPGA implementation. On the other hand, we give conservative estimations of the resource utilization on the Automata Processor for different datasets. By comparing the resource utilizations of FPGA and Automata Processor, we can have the following conclusions. First, the resource utilization of FPGA is much larger than that of Automata Processor if there are no counters and boolean elements. Second, since the numbers of counters and boolean elements are limited on Automata Processor, the resource utilizations of FPGA are smaller than the resource utilizations of Automata Processor for both 1DCA and 2DCA designs. In addition, because we can conduct pattern matching for 8~48 input streams concurrently on the Automata Processor, the Automata Processor is preferred in the case of multiple input streams. Furthermore, as a memory-based solution, the Automata Processor provides better reconfigurability than FPGA.

6.1 Future Work

First, Micron's Automata Processor has been announced in November 2013, but is not yet available for experimentations. Future work should focus on conducting various experiments on this platform. We can use different benchmarks to evaluate the performance and efficiency of automata-based tasks on the FPGA versus on the Automata Processor, providing insight into which technology is best suited for automata-based computations under various circumstances.

Second, hybrid logic-memory based designs for faster reconfiguration can be implemented for FPGA. Hybrid-FA [3] is a particular class of NFA that consists of multiple DFA connected in a hierarchical fashion. We will investigate the deployment of these automata on FPGA. For example, a possible implementation can use the one-hot encoding scheme [13] to represent interconnections among DFAs, and storing the DFA states in memory. More generally, given an FPGA, one can imagine partitioning the design into memory-based and logic-based components. A general way to effectively perform this partitioning on FPGA can be explored.

Finally, large datasets lead to the need for FA partitioning and judicious partitioning is required to efficiently leverage the on-board resources. FPGA-based designs require efficient routing among multiple hardware partitions or memory banks. FA partitioning can lead to a limited degree of state replication and parallel operating on different partitions. Therefore, efficient automata partitioning schemes can be explored in the future work.

REFERENCES

[1] V. Aho and M. J. Corasick. Efficient String Matching: An Aid to Bibliographic Search. In Communication of ACM, 18 (6): 333–340, June 1975.

[2] A. Andrew, “Game of Life Cellular Automata,” Springer, 2012(ISBN 978-1-84996-216-2).

[3] M. Becchi, and P. Crowley. A Hybrid Finite Automaton for Practical Deep Packet Inspection. In Proceedings of ACM CoNEXT, December 2007.

[4] M. Becchi, and P. Crowley, “An improved algorithm to accelerate regular expression evaluation,” in Proc. of ANCS 2007.

[5] M. Becchi, M. Franklin, and P. Crowley, “A workload for evaluating deep packet inspection architectures,” in Proceedings of the IEEE International Symposium on Workload Characterization, Seattle, Washington, 2008, pp. 79-89.

[6] M. Becchi, and P. Crowley, “Efficient regular expression evaluation: theory to practice,” in Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems, San Jose, California, 2008, pp. 50-59.

[7] M. Becchi, and P. Crowley, “Extending finite automata to efficiently match Perl-compatible regular expressions,” in Proceedings of the 2008 ACM CoNEXT Conference, Madrid, Spain, 2008, pp. 1-12.

[8] B. C. Brodie, D. E. Taylor, and R. K. Cytron, “A Scalable Architecture For High-Throughput Regular-Expression Pattern Matching,” in Proceedings of the 33rd annual international symposium on Computer Architecture, 2006, pp. 191-202.

[9] C. R. Clark, and D. E. Schimmel, “Efficient Reconfigurable Logic Circuits for Matching Complex Network Intrusion Detection Patterns,” in Proceedings of the 13th International Field Programmable Logic and Application Conference Lisbon, Portugal, 2003.

[10] CUDA. http://www.nvidia.com/object/cuda_home_new.html.

[11] P. Dlugosch, D. Brown, P. Glendenning et al., “An Efficient and ScalableSemiconductor Architecture for Parallel Automata Processing,” IEEE Transactions on Parallel and Distributed Systems, 2013.

[12] D. Ficara, S. Giordano, G. Procissi et al., “An improved DFA for fast regular expression matching,” SIGCOMM Comput. Commun. Rev., vol. 38, no. 5, pp. 29-40, 2008.

[13] R. W. Floyd and J. D. Ullman. The Compilation of Regular Expressions into

Integrated Circuits. In Journal of ACM, vol. 29, no. 3, pp 603-622, July 1982.

[14] J. Hopcroft, R. Motwani, and J. Ullman, Introduction to Automata Theory, Languages, and Computation: Addison Wesley, 1979.

[15] S. Kumar, J. Turner, and J. Williams, “Advanced algorithms for fast and scalable deep packet inspection,” in Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems, San Jose, California, USA, 2006, pp. 81-92.

[16] S. Kumar, S. Dharmapurikar, F. Yu et al., “Algorithms to accelerate multiple regular expressions matching for deep packet inspection,” in Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications, Pisa, Italy, 2006, pp. 339-350.

[17] S. Kumar, B. Chandrasekaran, J. Turner et al., “Curing regular expressions matching algorithms from insomnia, amnesia, and acalculia,” in Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems, Orlando, Florida, USA, 2007, pp. 155-164.

[18] C. Maxfield, “The Design Warrior's Guide to FPGAs: Devices, Tools and Flows,” Newnes, 2004 (ISBN: 0-7506-7604-3).

[19] R. McNaughton and H. Yamada. Regular Expressions and State Graphs for Automata. In IEEE Transactions on Electronic Computers, EC-9(1), pp. 39–47, 1960.

[20] NVIDIA, “NVIDIA’s Next Generation CUDA Compute Architecture: Fermi,” http://nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf, 2009.

[21] OpenMP. http://openmp.org/wp/.

[22] POSIX threads. https://computing.llnl.gov/tutorials/pthreads/.

[23] R. Sidhu, and V. K. Prasanna, “Fast Regular Expression Matching Using FPGAs,” in Proceedings of the the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2001, pp. 227-238.

[24] R. Smith, C. Estan, S. Jha et al., “Deflating the big bang: fast and scalable deep packet inspection with extended finite automata,” in Proceedings of the ACM SIGCOMM 2008 conference on Data communication, Seattle, WA, USA, 2008, pp. 207-218.

[25] R. Smith, C. Estan, and S. Jha, “XFA: Faster Signature Matching With Extended Automata,” in IEEE Symposium on Security and Privacy, 2008.

[26] SNORT NIDS. http://www.snort.org.

[27] I. Sourdis, J. Bispo, J. M. P. Cardoso et al., “Regular Expression Matching in Reconfigurable Hardware,” Signal Processing Systems, vol. 51, no. 1, pp. 99-121, 2008.

[28] S. Wolfram, Statistical Mechanics of Cellular Automata, Reviews of Modern Physics, volume 55, pages 601-644 (July 1983).

[29] Xilinx ISE Software: http://www.xilinx.com/products/design-tools/ise-design-suite/ise-webpack.htm.

[30] Xilinx Virtex-2P: http://www.xilinx.com/support/index.html/content/xilinx/en/supportNav/silicon_devices/mature_and_discontinued_products/virtex-ii_pro.html.

[31] Xilinx Virtex-4: http://www.xilinx.com/support/index.html/content/xilinx/en/supportNav/silicon_devices/fpga/virtex-4.html.

[32] Xilinx Virtex-5: http://www.xilinx.com/support/index.html/content/xilinx/en/supportNav/silicon_devices/fpga/virtex-5.html.

[33] F. Yu, Z. Chen, Y. Diao et al., “Fast and memory-efficient regular expression matching for deep packet inspection,” in Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems, San Jose, California, USA, 2006, pp. 93-102.

[34] X. Yu and M. Becchi, “GPU Acceleration of Regular Expression Matching for Large Datasets: Exploring the Implementation Space”. In Proc. of CF 2013.

VITA

Xiang Wang

EDUCATION

University of Missouri - Columbia Columbia, MO August 2012 to present

Master of Science in Computer Engineering

Research Interests:

- High-performance and parallel computer architectures, GPUs and FPGAs.
- Networking systems architectures, network security and deep packet inspection.

Shanghai University Shanghai, China August 2008 to July 2012

Bachelor of Science in Electrical Engineering

EXPERIENCE

Intern: **Micron Technology** May 2013 - August 2013 (3 months) San Jose, CA

Ran emulator and hardware tests for Micron’s Automata Processor. Used synthetic regular expression generation and synthetic traffic generation tools in network security for benchmark tests.

Graduate Research: **GPU and FPGA acceleration of regular expression matching**

MS Thesis Advisor: Dr. Michela Becchi University of Missouri – Columbia

Implemented DFA-based search engine on GPUs for regular expression matching. Implemented DFA compression algorithm and regular expression clustering algorithms to further optimize regular expression matching on GPUs. Converted both classical NFA and Automata Networks (an extension of NFA designed for Micron Technology's Automata Processor) to the logic representations on FPGA and conduct pattern matching on real and synthetic datasets.

Graduate Research: **Design of Runtime Technologies to Enable GPUs in HPC Clusters**

Graduate Research Advisor: Dr. Michela Becchi University of Missouri – Columbia

Created a benchmark generator for hybrid MPI-CUDA applications. Automatically generated a set of representative programs with different communication and computation patterns.

Capstone Project: **E-Menu System Design** University of Missouri – Columbia

Developed an android application to take restaurant orders with a Java program for PC terminals to receive those orders. Data communication was performed via WIFI, and passwords were transmitted through Zigbee.

PUBLICATIONS

K. Sajjapongse, X. Wang and M. Becchi, "A Preemption-based Runtime to Efficiently Schedule Multi-process Applications on Heterogeneous Clusters with GPUs," in Proc. of the 22nd Int'l ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC 2013), New York, NY, June 2013. [*acceptance rate: 15.3%*]

X. Wang, X. Yu and M. Becchi, "Evaluating Different Automata Representations for High-Speed Regular Expression Matching on GPUs", under submission to IEEE Transactions on Parallel and Distributed Systems

COMPUTER SKILLS

Programming languages: C, C++, Matlab, Java, Verilog, VHDL, Linux Shell Scripts

Operating systems: Linux, Windows

Parallel Programming models and libraries: POSIX Threads, MPI, CUDA, OpenMP