DEEP PACKET INSPECTION ON LARGE DATASETS: ALGORITHMIC AND PARALLELIZATION TECHNIQUES FOR ACCELERATING REGULAR EXPRESSION MATCHING ON MANY-CORE PROCESSORS

A Thesis

presented to

the Faculty of the Graduate School

at University of Missouri

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

---

by

XIAODONG YU

Dr. Michela Becchi, Advisor

JULY 2013

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

DEEP PACKET INSPECTION ON LARGE DATASETS: ALGORITHMIC AND PARALLELIZATION TECHNIQUES FOR ACCELERATING REGULAR EXPRESSION MATCHING ON MANY-CORE PROCESSORS

presented by

Xiaodong Yu

a candidate for the degree of

Master of Science

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

Dr. Michela Becchi, Assistant Professor, Dept. of Electrical and Computer Eng.

Dr. Guilherme DeSouza, Associate Professor, Dept. of Electrical and Computer Eng.

Dr.William Harrison, Associate Professor, Dept. of Computer Science

ACKNOWLEDGEMENTS

First and foremost, I am truly thankful to my advisor, Dr. Michela Becchi. Her limitless support enabled me to understand and accomplish this research. Without her patient guidance, timely help, and resolute dedication, this thesis could not have been completed. I would like also to express my sincere gratitude to Dr. Guilherme DeSouza and Dr. William Harrison for their assistance reviewing this manuscript.

Finally, I want to express my sincere thanks to my family and friends. This work is dedicated to my parents, Sanmao Yu and Junhui Song, who have given their complete love to me through my life and study. Their encouragement has given me great confidence in pursuing and finishing my Master degree in the U.S.

This work has been supported by National Science Foundation award CNS-1216756 and by equipment donations from Nvidia Corporation.

TABLE OF CONTENTS

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

DEEP PACKET INSPECTION ON LARGE DATASETS: ALGORITHMIC AND PARALLELIZATION TECHNIQUES FOR ACCELERATING REGULAR EXPRESSION MATCHING ON MANY-CORE PROCESSORS

Xiaodong Yu

Dr. Michela Becchi, Advisor

ABSTRACT

Regular expression matching is a central task in several networking (and search) applications and has been accelerated on a variety of parallel architectures, including general purpose multi-core processors, network processors, field programmable gate arrays, ASIC- and TCAM-based systems. All of these solutions are based on finite automata (either in deterministic or non-deterministic form), and mostly focus on effective memory representations for such automata. More recently, a handful of proposals have exploited the parallelism intrinsic in regular expression matching (i.e., coarse-grained packet-level parallelism and fine-grained data structure parallelism) to propose efficient regex matching designs for GPUs. However, most GPU solutions aim at achieving good performance on small datasets, which are far less complex and problematic than those used in real-world applications.

In this work, we provide a more comprehensive study of regular expression

matching on GPUs. To this end, we consider datasets of practical size and complexity and explore advantages and limitations of different automata representations and of various GPU implementation techniques. Our goal is not to show optimal speedup on specific datasets, but to highlight advantages and disadvantages of the GPU hardware in supporting state-of-the-art automata representations and encoding schemes, schemes which have been broadly adopted on other parallel memory-based platforms.

CHAPTER 1

INTRODUCTION

Regular expression matching is an important task in several application domains (bibliographical search, networking, and bioinformatics) and has received particular consideration in the context of deep packet inspection.

Deep packet inspection (DPI) is a fundamental networking operation, employed most notably at the core of network intrusion detection systems (NIDS). Some well-known open-source applications, such as Snort and Bro, fall into this category; in addition, all major networking companies are offering their own network intrusion detection solutions (e.g., security appliances from Cisco, Juniper Networks and Huawei Technologies).

A traditional form of deep packet inspection consists of searching the packet payload against a set of patterns. In NIDS, every pattern represents a signature of malicious traffic. As such, the payload of incoming packets is inspected against all available signatures, and a match triggers pre-defined actions on the interested packets.

Because of their expressive power, which can cover a wide variety of pattern

signatures [1-3], regular expressions have been adopted in pattern sets used in both industry and academia. In recent years, datasets used in practical systems have increased in both size and complexity: as of December 2011, over eleven thousand rules from the widely used Snort contain Perl-compatible regular expressions.

To meet the requirements of networking applications, a regular expression matching engine must both allow parallel search over multiple patterns and provide worst-case guarantees as to the processing time. An unbounded processing time would in fact open the way to algorithmic and denial of service attacks.

To allow multi-pattern search, current implementations represent the pattern-set through finite automata (FA) [4], either in their deterministic or in their non-deterministic form (DFA and NFA, respectively). The matching operation is then equivalent to a FA traversal guided by the content of the input stream. Worst-case guarantees can be met by bounding the amount of per character processing. 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 increases, limiting the size of the automaton becomes challenging. As a result, 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.

Previous work [5-18] has focused on accelerating regular expression matching on a variety of parallel architectures: general purpose multi-core processors, network

processors, FPGAs, ASIC- and TCAM-based systems. In all these proposals, particular attention has been paid to providing efficient logic- and memory-based representations of the underlying automata (namely, DFAs, NFAs and equivalent abstractions).

Because of their massive parallelism and computational power, in recent years GPUs have been considered a viable platform for this application [19-23]. However, existing work has mostly evaluated specific solutions on small datasets, consisting of a few tens of patterns.

In general, there is disconnect between the richness of proposals (in terms of automata and their memory representation) emerged in the context of other memory-centric architectures [5-15, 24], and their evaluation on GPU platforms, especially on large and complex datasets which are relevant to today's applications. Besides leaving the suitability of GPUs to real-world scenarios unclear, this lacuna tends to allow proposals focusing on trivial datasets with little practical relevance and those present automata abstractions that appear innovative but are essentially equivalent to existing solutions.

1.1 Our Contributions

In this work, we target the problem that mentioned above. Our contributions can be summarized as follows.

- We present an extensive GPU-based evaluation of different automata

representations on datasets of practical size and complexity.

- We show three simple schemes to avoid some of the limitations of a recent NFA-based design [22].
- We discuss the impact of state-of-the-art memory compression schemes [5, 10] on DFA solutions.
- We evaluate the use of software managed caches on GPU.

1.2 Thesis Organization

The rest of this thesis is organized as follows. In Chapter 2, we provide some more background and discuss related work in the context of regular expression matching. In Chapter 3, we present different regular expression matching engine designs for GPUs based on NFAs, DFAs and Hybrid-Fas including the optimizations and caching scheme. In Chapter 4, we provide an experimental evaluation of all the proposed schemes on a variety of pattern-sets, both real and synthetic. In Chapter 5, we conclude our discussion.

CHAPTER 2

BACKGROUND AND RELATED WORK

2.1 Background on Regular Expression Matching

A regular expression, often called pattern, is a compact representation of a set of strings. Regular expressions provide a flexible text processing mechanism: an operation common to many applications (e.g. bibliographic search, deep packet inspection, bio-sequence analysis) consists of analyzing large corpora (or texts) to find the occurrence of patterns of interest.

Formally, a regular expression over an alphabet $\sum$ is defined as follows:

1. Ø (*empty set*) is a regular expression corresponding to the empty language Ø.
2. ε (*empty string*) is a regular expression denoting the set containing only the "empty" string, which has no character at all.
3. For each symbol $a \in \sum$, a is a regular expression (*literal character*) corresponding to the language $\{a\}$.
4. For any regular expressions *r* and *s* over $\sum$, corresponding to the languages L_r

and L_s respectively, each of the following is a regular expression corresponding to the language indicated:

a. *concatenation*: (*rs*) corresponding to the language L_rL_s

b. *alternation*: (*r*+*s*) corresponding to $L_r \cup L_s$

c. *Kleene star:* r^* corresponding to the language L_{r^*}

5. Only those "formulas" that can be produced by the application of rules 1-4 are regular expressions over $\sum$.

Regular expression matching is implemented using finite automata. A finite automaton consists of: a finite set of states; a finite input alphabet that indicates the allowed symbols; rules for transitioning from one state to another depending upon the input symbol; a start state; a finite set of accepting states. Mathematically, a finite automaton is typical defined as a 5-tuple ($Q, \Sigma, \delta, q_0, F$) so defined:

1. Q is a finite set of states
2. Σ is a finite set of symbols, or alphabet
3. δ : is the state transition function
4. $q_0 \in Q$ is the start (or initial) state
5. $F \subseteq Q$ is the set of accepting (or final) states.

Finite automata can be deterministic or non-deterministic. In Deterministic Finite Automata (DFAs), the transition function δ is of the kind $Q \times \Sigma \rightarrow Q$: given an active state and an input symbol, δ leads always to a single active state. In Non-deterministic Finite Automata (NFAs), the transition function δ is of the kind $Q \times \Sigma \rightarrow P(Q)$: given an

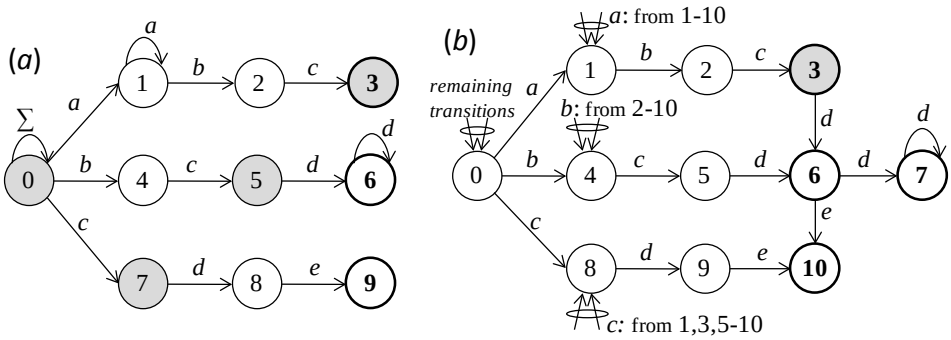


Figure 1: (a) NFA and (b) DFA accepting regular expressions a+bc, bcd+ and cde. Accepting states are bold. States active after processing text aabc are colored gray. In the NFA, ∑ represents the whole alphabet. In the DFA, state 4 has an incoming transition on character b from all states except 1 (incoming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

active state and an input symbol, it can activate a set of states (that may be empty).

The exploration space for finite automata 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. In particular, NFAs have a limited size but can require expensive per-character processing, whereas DFAs offer limited per-character processing at the cost of a possibly large automaton. To provide intuition regarding this fact, in Figure 1 we show the NFA and DFA accepting 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 (∑). Each DFA state corresponds to a set of NFA states that can be simultaneously active [4]; therefore, the number of states in a DFA equivalent to an N-state NFA can potentially

be 2^N. In reality, previous work [6, 9, 12, 14, 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). Since each DFA state corresponds to a set of simultaneously active NFA states, DFAs ensure minimal per-character processing (only one state transition is taken for each input character).

2.2 Implementation Approaches

From an implementation perspective, existing regular expression matching engines can be classified into two categories: memory-based [5-15, 24] and logic-based [7, 16-18]. 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 (and have been) deployed on various parallel platforms: general purpose multi-core processors, network processors, ASICs, FPGAs, and GPUs; logic-based implementations typically target FPGAs. Of course, for the logic-based approaches, updates in the pattern-set require the underlying platform to be reprogrammed. In a memory-based implementation, design goals are the minimization of the memory size needed to store the automaton and of the memory bandwidth needed to operate it. Similarly, in a logic-based implementation the design should aim at minimizing the logic utilization while allowing fast operation (that is, a high clock frequency). Memory-based solutions have been recently adapted and extended to TCAM

implementations [25-27].

Existing proposals targeting DFA-based, memory-centric solutions have focused on two aspects: (i) designing compression mechanisms aimed at minimizing the DFA memory footprint; and (ii) devising novel automata to be used as an alternative to DFAs in case of state explosion. Alphabet reduction [5, 13, 28], run-length encoding [13], default transition compression [5, 10], and delta-FAs [15] are generally applicable mechanisms falling into the first category, whereas multiple-DFAs [12, 13], hybrid-FAs [6, 24], history-based-FAs [9] and XFAs [14] fall into the second one. All DFA compression schemes leverage the transition redundancy that characterizes DFAs describing practical datasets. 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 (or input streams).

2.3 Introduction to Graphics Processing Units and GPU-based Engines

In recent years Graphics Processing Units (GPUs) have been widely used to accelerate a variety of scientific applications [29-31]. Most proposals have target NVIDIA GPUs, whose programmability has greatly improved since the advent of CUDA [32]. The main architectural traits of these devices can be summarized as follows.

NVIDIA GPUs comprise a set of Streaming Multiprocessors (SMs), each of them

containing a set of simple in-order cores. These in-order cores execute the instructions in a SIMD manner. GPUs have a heterogeneous memory organization consisting of high latency global memory, low latency read-only constant memory, low-latency read-write shared memory, and texture memory. GPUs adopting the Fermi architecture, such as those used in this work, are also equipped with a two-level cache hierarchy. Judicious use of the memory hierarchy and of the available memory bandwidth is essential to achieve good performance.

With CUDA, the computation is organized in a hierarchical fashion, wherein threads are grouped into thread blocks. Each thread-block is mapped onto a different SM, while different threads in that block are mapped to simple cores and executed in SIMD units, called warps. Threads within the same block can communicate using shared memory, whereas threads from different thread blocks are fully independent. Therefore, CUDA exposes to the programmer two degrees of parallelism: fine-grained parallelism within a thread block and coarse-grained parallelism across multiple thread blocks. Branches are allowed on GPU through the use of hardware masking. In the presence of branch divergence within a warp, both paths of the control flow operation are in principle executed by all CUDA cores. Therefore, the presence of branch divergence within a warps leads to core underutilization and must be minimized to achieve good performance.

Recent work [33] has considered exploiting the GPU's massive hardware parallelism and high-bandwidth memory system in order to implement

high-throughput networking operations. In particular, a handful of proposals [19-23] have looked at accelerating regular expression matching on GPU platforms. Most of these proposals use the coarse-grained block-level parallelism offered by these devices to support packet- (or flow-) level parallelism intrinsic in networking applications.

Gnort [19, 20], proposed by Vasiliadis et al, represents an effort to port Snort IDS to GPUs. To avoid dealing with the state explosion problem, the authors process on GPU only a portion of the dataset consisting of regular expressions that can be compiled into a DFA, leaving the rest in NFA form on CPU for separate processing. As a result, this proposal speeds up the average case, but does not address malicious and worst case traffic. In Gnort, the DFA is represented on GPU memory uncompressed, and parallelism is exploited only at the packet level (i.e., no data structure parallelism is exploited to further speed up the operation).

Smith et al [21] ported their XFA proposed data structure [14] to GPUs, and compared the performance achieved by an XFA- and a DFA-based solution on datasets consisting of 31-96 regular expressions. They showed that a G80 GPU can achieve a 10-11X speedup over a Pentium 4, and that, because of the more regular nature of the underlying computation, on GPU platforms DFAs are slightly preferable to XFAs. It must be noticed that the XFA solution is suited to specific classes of regular expressions: those that can be broken into non-overlapping sub-patterns separated by ".*" terms. However, these automata cannot be directly applied to

regular expressions containing overlapping sub-patterns or $[\hat{}c_1 \ldots c_k]^*$ terms followed by sub-pattern containing any of the c_1, …, c_k characters.

More recently, Cascarano et al [22] proposed iNFAnt, a NFA-based regex matching engine on GPUs. Since state explosion occurs only when converting NFAs to DFAs, iNFAnt is the first solution that can be easily applied to rule-sets of arbitrary size and complexity. In fact, this work is, to our knowledge, the only GPU-oriented proposal which presents an evaluation on large, real-world datasets (from 120 to 543 regular expressions). The main disadvantage of iNFAnt is its unpredictable performance and its poor worst-case behavior. In Section 3.2, we will this solution in more detail.

Zu et al [23] proposed a GPU design which aims to overcome the limitations of iNFAnt. The main idea is to cluster states into compatibility groups, so that states within the same compatibility group cannot be active at the same time. The main limitation of this method is the following. The computation of compatibility groups requires the exploration of all possible NFA activations (this fact stems from the definition of compatibility groups itself). This, in turn, is equivalent to subset construction (i.e., NFA to DFA transformation). As highlighted in previous work [6, 9, 12, 14, 24], this operation, even if theoretically possible, is practically feasible only on small or simple rule-sets that do not incur the state explosion problem. Not surprisingly, the evaluation proposed in [23] is limited to datasets consisting of 16-36 regular expressions. Further, the transition tables of these datasets are characterized by

a number of distinct transitions per character per entry ≤ 4 (although they are not systematically built to respect this constraint). This proposal is conceptually very similar to representing each rule-set through four DFAs, which is also feasible only on small and relatively simple datasets. As a consequence, we believe that the comparison with iNFAnt is unfair. A comparison with a pure DFA-based approach would be more appropriate. However, given its nature, the proposal in [23] is likely to provide performance very similar to a pure DFA-based solution.

In this work, we evaluate GPU designs on practical datasets (with size and complexity comparable to those used in [22]). In contrast to [23], our goal is not to show optimal speedup of a given solution on a specific kind of rule-set. Instead, we want to provide a comprehensive evaluation of automata representations, memory encoding and compression schemes that are commonly used in other memory-centric platforms. We hope that this analysis will help users to make an informed selection among the plethora of existing algorithmic and data structure proposals.

CHAPTER 3

OUR GPU IMPLEMENTATION

In all our regular expression engine designs, the FA traversal, which is the core of the pattern matching process, is fully implemented in a GPU kernel. The implementations presented in this proposal differ in the automaton used, and, as a consequence, in the memory data structures stored on GPU and in the FA traversal kernel. However, the CPU code is the same across all the proposed implementations. We first describe the CPU code and the computation common to all implementations (Section 3.1), and then our NFA, DFA and HFA traversal kernels (Section 3.2, 3.3 and 3.4 respectively).

3.1 General Design

Like previous proposals, our regular expression matching engine supports multiple packet-flows (that is, multiple input streams) and maps each of them onto a different thread-block. In other words, we support packet-level parallelism by using the different SMs available on the GPU. The size of the packets (P_{SIZE}) is configurable and set to 64KB in all our experiments. The number of packet-flow processed in

parallel (N_{PF}) is also configurable: if it exceeds the number of SMs, multiple packet-flows will be mapped onto the same SM. Packet-flows handled concurrently may not necessarily have the same size: we use an array of flow identifiers to keep track of the mapping between the packets and the corresponding packet-flows. When a packet-flow is fully processed, the corresponding thread-block is assigned to a new flow.

The FA is transferred from CPU to GPU only once, at the beginning of the execution. Then, the control-flow on CPU consists of a main loop. The operations performed in each iteration are the following. First, N_{PF} packets – one per packet-flow – are transferred from CPU to GPU and stored contiguously on the GPU global memory. Second, the FA traversal kernel is invoked, thus triggering the regex matching process on GPU. The result of the matching operation is transferred from GPU to CPU at the end of the flow-traversal. The state information, which must be preserved in ordered to detect matches that occur across multiple packets, can be kept on GPU and does not need to be transferred back to CPU. However, such information must be reset before starting to process a new packet-flow.

We use double buffering in order to hide the packet transfer time between CPU and GPU. To this end, on GPU we allocate two buffers of N_{PF} packets each: one buffer stores the packets to be processed in the current iteration, and the other the ones to be processed in the next iteration. The FA-traversal corresponding to iteration i can overlap with the packet transfer related to iteration i+1. The function of the two

buffers is swapped from iteration to iteration.

3.2 NFA-based Engines

The advantage of NFA-based solutions is that they allow a compact representation for datasets of arbitrary size and complexity. The pattern-set to NFA mapping is not one-to-one: it is possible to construct many NFAs accepting the same pattern-set (and, in fact, a DFA can be seen as a special case NFA). In particular, it is always possible to build an NFA with a number of states which is less than or equal to that of characters in the pattern-set. However, state explosion may occur when transforming an NFA into DFA, since each DFA state corresponds to the set of NFA states which can be active in parallel. In this section, we consider pure NFA solutions, that is, solutions that are applicable to arbitrary pattern-sets (and that rely on NFAs built to have a minimal number of states). As anticipated in Section 2, the most efficient GPU-based NFA proposal is, to our knowledge, the iNFAnt design [22]. In fact, while [23] provides better performance than iNFAnt, it implicitly requires a full analysis of the potential NFA activations to compute the state compatibility groups. Such analysis is equivalent to an NFA-to-DFA transformation (feasible only on small or simple datasets). Therefore, the design proposed in [23] is not applicable to arbitrary NFAs. Such solution is almost equivalent to using a set of DFAs (specifically, four DFAs for the datasets and the settings used in [23]).

3.2.1 iNFAnt

In the *iNFAnt* proposal, the transition table is encoded using a *symbol-first* representation: transitions are represented through a list of (*source, destination*) pairs sorted by their triggering symbol, whereby *source* and *destination* are 16-bit state identifiers. An ancillary data structure records, for each symbol, the first transition within the transition list. Persistent states (i.e., states with a self-transition on every character of the alphabet) are handled separately using a state vector. These states, once activated, will remain active for the whole NFA traversal. The iNFAnt kernel operates as shown in the pseudo-code below, which is adapted from [22]. For readability, in all the pseudo-code reported in this paper, we omit representing the matching operation occurring on accepting states.

Besides the persistent state vector ($persistent_{sv}$), *iNFAnt* uses two additional state vectors: $current_{sv}$ and $future_{sv}$, which store the current and the future set of active states. All state vectors are represented as bit-vectors, and stored in shared memory. After initialization (line 1), the kernel iterates over the characters in the input stream (loop at line 2). The bulk of the processing starts at line 5, after character *c* is retrieved from the input buffer (lines 3-4). First, the future state vector is updated to include the active persistent states (line 5). Second, the transitions on input *c* are selected (lines 6-8), and the ones originating from active states cause $future_{sv}$ to be updated (lines 9-10). Finally, $current_{sv}$ is updated to the value of $future_{sv}$ (line 11). Underlined statements represent barrier synchronization points. As can be seen (line 10), an

additional synchronization is required to allow atomic update of the future state vector.

---

kernel iNFAnt

```
1:  current_sv ← initial_sv
2:  while !input.empty do
3:      c ← input.first
4:      input ← input.tail
5:      future_sv ← current_sv & persistent_sv
6:      while a transition on c is pending do
7:          src ← transition source
8:          dst ← transition destination
9:          if current_sv[src] is set then
10:             atomicSet(future_sv, dst)
11:     current_sv ← future_sv
end;
```

Besides thread-block parallelism at the packet level, this kernel exploits thread-level parallelism in several points. First, the state vector updates at lines 1, 5 and 11 are executed in parallel by all threads. In particular, consecutive threads access consecutive memory words, thus ensuring conflict-free shared memory accesses. Second, the transition processing loop (lines 6-10) is also parallelized over the threads in a block, and the layout of the transition table allows coalesced global memory accesses during this operation. Note that the number of threads concurrently accessing the state vectors and the transition list is equal to the block size: in case of large NFAs, multiple iterations are required to perform a full scan of these arrays.

Since the state vectors are represented as bit-vectors and the transition table contains 16-bit state identifiers, a translation between a bit-vector and a short integer

state identifier representation is required in lines 9 (for *src*) and 10 (for *dst*). This operation is relatively costly.

3.2.2 Our optimized NFA-based design

We note that *iNFAnt* presents the following inefficiency. The transition processing loop at line 6 must scan all NFA transitions on a given symbol *c*. On large NFAs, such list can be very long and may require a large number of thread-block iterations. In every iteration, global memory is accessed to retrieve the transition information (line 6), and several translations of state identifiers into bit-vectors are required (line 9). The latter operation is performed even if the corresponding states are inactive. Therefore, with the exception of instruction 10, the cost of the loop at line 6 is *independent of the size of the current state vector*. This makes iNFAnt's average-case processing time comparable to its worst-case. In the worst-case, instruction 10 is executed on all retrieved transitions.

The average-case processing can be improved by taking the following observations into consideration.

1. In most traversal steps, only a minority of the NFA states are active.
2. NFA states can be easily clustered into groups of states that cannot be active at the same time.

The second observation is also exploited by [23]. Differently from [23], we do not want to fully explore the compatibility between NFA states, because this would require a full exploration of all possible NFA activations, which is impractical on

relatively large and complex datasets.

One of the advantages of iNFAnt is the simplicity of its implementation. We aim to introduce code modifications that maintain the simplicity of the original design. Ad-hoc solutions targeting too specific situations can originate divergence during execution, and end up being not beneficial in the average case.

Optimization 1: The first consideration can be exploited as follows. If transitions on the same character are sorted according to the source state identifier, and the largest active state identifier SID_{MAX} is known, then the execution of loop 6 can be terminated when the first transition with source identifier greater than SID_{MAX} is encountered. In case of large NFAs, this can significantly reduce the number of iterations required to process the transitions on the current input symbol. We implement this optimization as follows. First, we store SID_{MAX} in shared memory (and save it to global memory at the end of each packet). Second, we add the instruction *atomicMax(*SID_{MAX}*, dst)* to the *if*-statement at line 9. This updates SID_{MAX} whenever required (the update is performed in shared memory). Third, in implementing the loop at line 6, we check whether the last retrieved transition has source state identifier greater than SID_{MAX}, in which case we terminate the loop. More specifically, our implementation requires twos version of SID_{MAX}: a current and a future value. These variables are handled similarly to the current and future version of the active state vector.

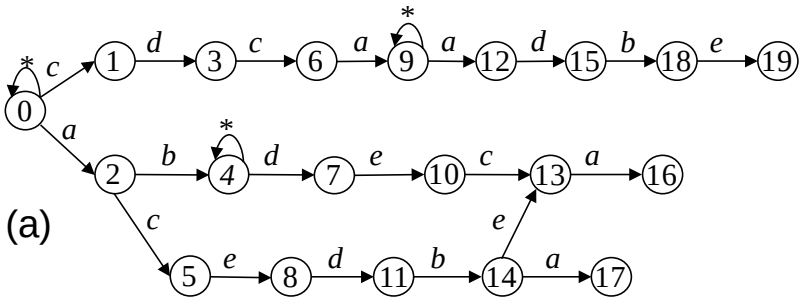
Optimization 2: The second observation can be exploited as follows. States can

be grouped according to their incoming transitions. We define two or more states as *compatible* if they can potentially be active at the same time. For example, states that share at least one incoming transition (i.e., the symbol on that transition) are compatible. We perform the following state clustering. We distinguish states with a single incoming transition and states with multiple incoming transitions. States with a single incoming transition on character c_i are grouped into $group_{ci}$. The other states are grouped into a special group, that we call $group_{overlap}$. Clearly, the following holds. (i) States belonging to the same group are compatible. (ii) States belonging to different $group_{ci}$ are incompatible. (iii) States belonging to $group_{overlap}$ are compatible with any other state. Compatibility groups will decrease in size (and increase in number) if a finer grained classification is made (for instance, if sequences of two or more incoming characters are considered in defining the compatibility groups). However, most of the NFA states have a single incoming transition, and we experimentally verified that a finer grained classification does not significantly affect the performance of our implementation.

We can take advantage of the above classification as follows. Loop 6 can be made more efficient by grouping transitions on the same input character according to the compatibility group of their source state. For simplicity, we will say that two transitions are compatible when their source states are such. At each iteration, rather than processing all transitions on the current input symbol, we can consider only the ones that are compatible with the current active states. Note that, because of our

definition of compatibility, all the states in the active set (that is, in $current_{sv}$) must be compatible: they must belong either to the same $group_{ci}$ or to the $group_{overlap}$. The compatibility group of the current active set depends on the previously processed input character c_P. To summarize, when processing the current input character c, the traversal can be limited to two subsets of the transitions on c: those in $group_{overlap}$ and those in $group_{cP}$. This can be easily achieved by sorting the transitions on the same character according to their compatibility group, and by using an ancillary array indicating the offset of each group within this hierarchical transition list.

Optimization 3: To combine the benefits of the two optimizations presented above, a proper *numbering scheme* for state identifiers should be adopted. We make three observations. First, persistent states and states with self-loops belong to the $group_{overlap}$. These states have a higher likelihood to be active during the traversal (and, once activated, they may be never or hardly deactivated). Second, states close to the NFA entry state are more likely to be traversed. Third, the benefits of optimization 1 will be higher if the state numbering scheme is such that states within the same compatibility group have similar identifiers. Given these considerations, we adopt the following state numbering scheme. First, we number the states according to a breadth-first traversal of the NFA. Second, we compute the compatibility groups. Third, we order the compatibility groups as follows. First, we consider the $group_{overlap}$. Then, we sort all $group_{ci}$ according to the average frequency of character c on publicly available packet traces. Finally, we assign the source identifiers in sequence starting



Orig. ID	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
New ID	0	10	4	13	1	11	12	14	17	2	18	15	5	3	8	16	6	7	9	19

(b)

	Orig. state ID	New state ID
overlap	0, 4, 9. 13	0, 1, 2, 3
a	2, 12, 16, 17	4, 5, 6, 7
b	14, 18	8, 9
c	1, 5, 6	10, 11, 12
d	3, 7, 11, 15	13, 14, 15, 16
e	8, 10, 19	17, 18, 19

(c)

Figure 2: (a) Sample NFA, (b) state renaming scheme, (c) state compatibility groups before and after state renaming.

from the first group. Within each group, we keep the priority dictated by the identifiers set in the first phase (thus preserving the breadth-first ordering of states within groups).

In Figure 2, we show an example. In particular, Figure 2(a) shows the original NFA. Figure 2(b) shows the state numbering assigned with breadth-first traversal (*Orig. ID*) and with the described state numbering scheme (*New ID*). Figure 2(c) shows the state compatibility groups, and reports the state identifiers in the original NFA of Figure 2(a), and those computed with our scheme. In Figure 3, we show the memory layout of the NFA in Figure 2. For readability, we show only the transitions on characters *a*, *b* and *c*. As can be seen, the transition list is hierarchical: first, transitions are sorted according to their input character; second, transitions on the

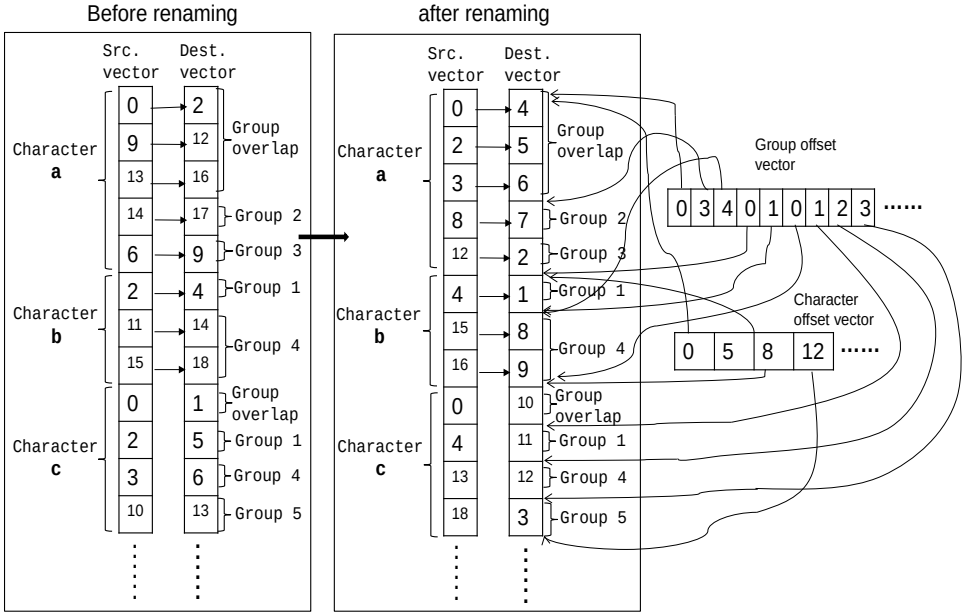


Figure 3: Memory layout of the transitions on characters a, b and c for the NFA of Figure 2.

same input character are sorted according to their compatibility group. As mentioned before, the current input character is used in the first-level selection, and the previously processed input character c_P is used in the second-level access.

3.3 DFA-based Engines

The main weakness of NFAs is their non-deterministic nature. During NFA traversal, the number of active states can vary from iteration to iteration; and so does the amount of work performed in different phases of the traversal. On the other hand, thanks to their deterministic nature, DFAs can offer predictable and bounded per character processing. As mentioned above, the presence of repetitions of wildcards and large character sets may lead to state explosion when transforming an NFA into

DFA, making a DFA solution impractical. DFAs are never a viable solution in the presence of unanchored regular expressions containing bounded repetitions of character sets [12, 24]. Becchi & Crowley [24] proposed a counting automaton to tackle this kind of patterns. In this work, we do not take counting constraints in consideration: they can, however, be supported through our NFA-base design. The remaining problematic patterns, namely unbounded repetitions of wildcards and large character sets, can be handled by grouping regular expressions into clusters and by generating multiple DFAs [12, 34], one foreach cluster. The pattern matching operation requires all these DFAs to be traversed on each input character, thereby causing memory bandwidth pressure.

There is no established optimal mechanism to cluster an arbitrary set of regular expressions. In this paper, we use the clustering algorithm proposed in [34]. The basic idea is the following: rather than statically determining the number of DFAs to generate, the user defines the maximum allowed DFA size. The generation process will try to cluster together as many regular expressions as possible so that the corresponding DFA size will not exceed the defined threshold. This is done by first attempting to generate a DFA accepting the whole pattern-set. If, during DFA generation, the maximum DFA size is reached, the regular expression set is automatically split in two. The algorithm proceeds recursively in a divide and conquer fashion, until all regular expressions have been processed. Typically this will lead to a small number of DFAs for small and simple datasets, and in a large number of DFAs

for large and complex ones. In our experiments we set the maximum DFA size to 64K states, to allow the use of 16-bit state identifiers.

A naïve way to implement a DFA-based regular expression matching engine is to treat multiple DFAs as a single NFA (with a constant active set size). In other words, iNFAnt can be used to support multiple DFAs. Given the large number of DFA states, however, this implementation would be very inefficient. A better design exploits the fact that each DFA state has one and only one outgoing transition on every input character.

3.3.1 Uncompressed DFA-based solution

The bulk of the traversal of a set of uncompressed DFAs is represented in the pseudo-code below. Again, we assume that different thread-blocks are dedicated to different packet-flows (or input streams). Thread-level parallelism is exploited at a DFA granularity: each thread processes a different DFA.

---

kernel uncompressed-DFA

1: $current_s \leftarrow initial_{sv}[tid]$
2: **while** *!input.empty* **do**
3: $c \leftarrow input.first$
4: $input \leftarrow input.tail$
5: $current_s \leftarrow state\_table[tid][current_s][c];$
6: $initial_{sv}[tid] \leftarrow current_s$

end

The underlying data structure consists of a bi-dimensional transition table per DFA (*state_table* in the pseudo-code), which stores the destination state identifier for

every (source state, input character) pair. State tables corresponding to different DFAs are laid out next to one another in global memory. Each thread stores the information of the current active state in a local register (*current*$_s$ variable), which is initialized from global memory and copied back to it at the beginning and at the end of each packet's processing (lines 1 and 6, respectively). In each iteration of the main loop (which starts at line 2), each thread performs a global memory access to query the state table on the input character (line 5). Note that such memory accesses are scattered and uncoalesced. The only way to hide the memory latency is to increase the global thread count by processing multiple packet-flows on the same SM. This is possible given the limited shared memory and register requirement of this simple DFA traversal kernel. Note that, in case of small datasets leading to a single or few DFAs, the code can be slightly modified so to make each thread-block process multiple packet-flows. For example, in the presence of 4 DFAs, a 32-thread block (sized to handle a full warp), can process 8 packet-flows in parallel. Since our design aims at evaluating generally applicable solutions on large and complex datasets, we do not consider this optimization in our experimental evaluation.

3.3.2 Compressed DFA-based solution

Uncompressed DFAs are characterized by one outgoing transition per character per state. In the presence of large alphabets and millions of states, this may lead to high memory requirements. This problem has been extensively studied. All existing DFA memory compression schemes exploit the transition redundancy inherent in

these automata. Perhaps the most effective DFA compression mechanism has been proposed in [10] and improved in [5]. The basic idea is to connect pair of states S_1 and S_2 characterized by transition commonality through non-consuming directed *default* transitions. Then, all transitions common to S_1 and S_2 can be safely removed from the source state S_1. This mechanism leads to a processing overhead due to default transition traversal: the algorithm proposed in [5] minimizes such overhead and ensures a worst-case $2N$ state traversals to process an input stream consisting of N characters. Other compression mechanisms, such as alphabet reduction [5, 13], are mostly ineffective on large datasets.

The pseudo-code below represents a kernel implementing multiple-DFA traversal in the presence of default transition compression. In a default transition-compressed DFA, each state has a default transition and a variable number of labeled transitions. Default transitions can be stored in a one-dimensional array (*default_tx*) with as many entries as DFA states. Labeled transitions can be represented through a list of (input character, destination state) pairs, and an ancillary data structure (*offset* array) indicating, for each state, the offset in the transition list. In our implementation, to optimize the memory usage, we store labeled transitions in a one-dimensional array (*labeled_tx*) of 32-bit elements. For each element, 8 bits are used to represent the ASCII input character, and the remaining bits to store the state identifier. Note that this allows supporting DFAs with up to 16M states (well above the 64K threshold used in this work). These data structures are stored in global memory; the only

exception being the frequently accessed initial state, which is stored in constant memory and accessed through the constant memory cache.

---

kernel compressed-DFA

```
1:  current_sv[tid.y] ← initial_sv[tid.y]
2:  idx[tid.y] ← 0
3:  while (idx[tid.y]!=PACKET_SIZE) do
4:      future_sv[tid.y] ← INVALID
5:      c ← input[idx[tid.y]]
6:      tx_offset ← offset[current_sv[tid.y]]
7:      while current states has unprocessed transitions
8:          symbol ← labeled_tx[tid.y][tx_offset][it_offset+tid.x].char
9:          dst    ← labeled_tx[tid.y][tx_offset][it_offset+tid.x].dst
10:         if (symbol = c)
11:             future_sv[tid.y] ← dst
12:             idx[tid.y]++
13:     if (future_sv[tid.y] = INVALID)
14:         future_sv[tid.y] ← default_tx[tid.y][current_sv[tid.y]]
15:     current_sv[tid.y] = future_sv[tid.y]
16: initial_sv[tid.y] ← current_sv[tid.y]
```

end

Thread-level parallelism is exploited in two ways: different threads process different DFAs and, within the same DFA, different labeled transitions. This is accomplished by using *bi-dimensional thread-blocks*. The logical mapping and partitioning is represented in Figure 4. As can be seen, different input streams are again mapped onto different thread-blocks. Threads within the same block are partitioned over multiple DFAs (using the *y*-dimension of the thread identifier). Within each partition, different threads process different labeled transitions (using the *x*-dimension of the thread identifier). The DFA data structures are laid out in memory next to one another.

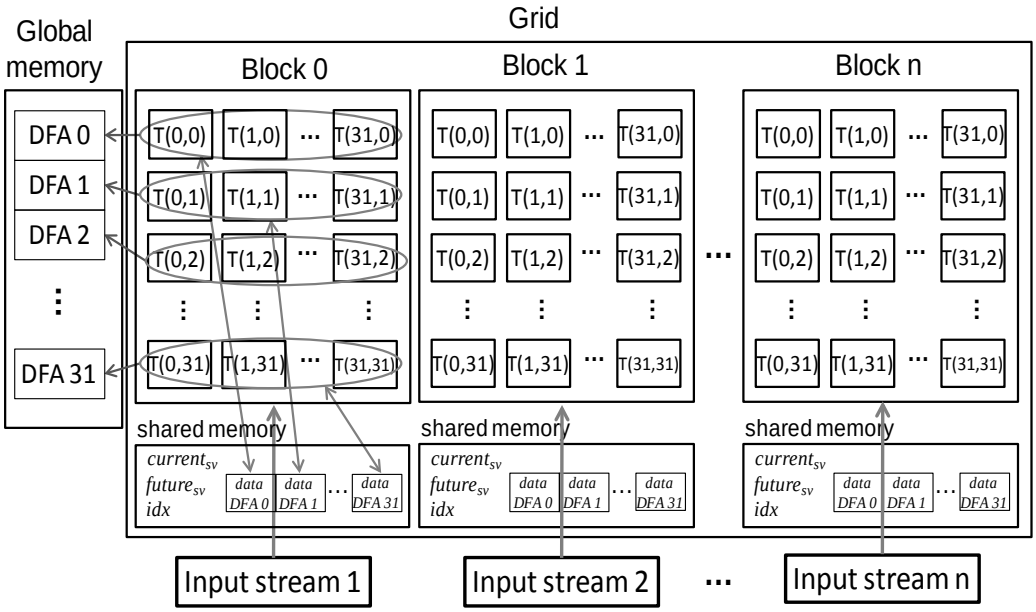


Figure 4: Exemplification of mapping of DFAs and input streams on the GPU in the compressed-DFA-based solution.

The data structures stored in shared memory are the following: (i) the active state vector $current_{sv}$, initialized from global memory and restored to it at the beginning and at the end of each packet's processing (lines 1 and 15, respectively); (ii) the future state vector $future_{sv}$, with the same use as in the NFA implementation; and (iii) the vector *idx*, containing a pointer to the next character to be processed. Each of these arrays has one entry per DFA, which must be shared along the *x*-dimension by all threads processing the same DFA. For efficiency, variables *c*, *tx_offset*, *symbol* and *dst* reside in (per-thread) registers.

The bulk of the processing in the main loop (starting at line 3) can be summarized as follows. In lines 7-12, the labeled transitions are fetched from global memory by different threads. The corresponding memory accesses are coalesced. If the number of labeled transitions is larger than the *x*-dimension of the thread-blocks, this operation

may require several iterations ("*it_offset*" represents the iteration offset). If a transition on the current input character is found (line 10), the future state vector is updated and the pointer *idx* is moved forward. Otherwise, the default transition is taken (lines 13-14).

As can be seen (lines 3 and 12), because of the use of default transitions, different DFAs may be processing different characters of the input stream at the same time. The use of the algorithm proposed in [5] minimizes the number of extra state traversals performed. However, default transitions may still lead to warp divergence and badly affect the performance. After default transition compression, more than 90% of the states are left with 4-5 labeled transitions. If the *x*-dimension of the block size is set to 32 (to equal the warp size), an iteration of the loop at line 7 is sufficient to process most input characters. However, such setting leads to warp underutilization.

3.3.3 Enhanced compressed DFA-based solution

We want to overcome the main limitations of our compressed-DFA design. Specifically, we want to reduce warp divergence and thread underutilization. Further, given that previous work [34] has shown that regular expression matching exhibits strong locality, we implement a software-managed caching scheme in shared memory.

To achieve these goals we reorganize the layout of the transition table in a more regular way. Specifically, we allow compressed states to consist of either 4 or 8 labeled transitions. All states with more than 8 transitions are represented in full and

processed via direct indexing. States with less than 4 or with more than 4 but less than 8 labeled transitions are padded (by duplicating some existing transitions), as shown in Figure 5(a). In addition, for each DFA, we reorder and renumber the states so to accommodate states of the same kind contiguously. Each memory region is identified by a *base_offset* variable, which stores the identifier of the first state in that region, and is stored in a register for efficient access. For example, in Figure 5(b), $base\_offset_4 = 1$, $base\_offset_8 = n+1$ and $base\_offset_{256}=n+m+1$.

With this layout, processing compressed states requires a maximum of 8 CUDA threads and always a single iteration of the loop at lines 7-12 in the pseudo-code above. We modify the thread-block organization so to have 8 threads along the *x*-dimension; we keep the size of the *y*-dimension equal to the number of DFAs. Each warp can now process 4 (rather than a single) DFAs. In this design, warp divergence can take place when, on a given input character, some of the DFAs processed by the same warp follow a default while others follow a labeled transition. However, the more uniform state layout and the reduction of the *x*-dimension of the block size allow higher thread utilization.

The memory layout described above facilitates our cache design. Since, on Fermi GPUs, memory transactions are 128 bytes wide, we set the size of cache blocks to be a multiple of 128 bytes. For illustration, in Figure 5(b) we assume 128-byte blocks: each block contains either 8 states with 4 transitions each, or 4 states with 8 transitions each. We do not cache uncompressed states. We need a tagging mechanism

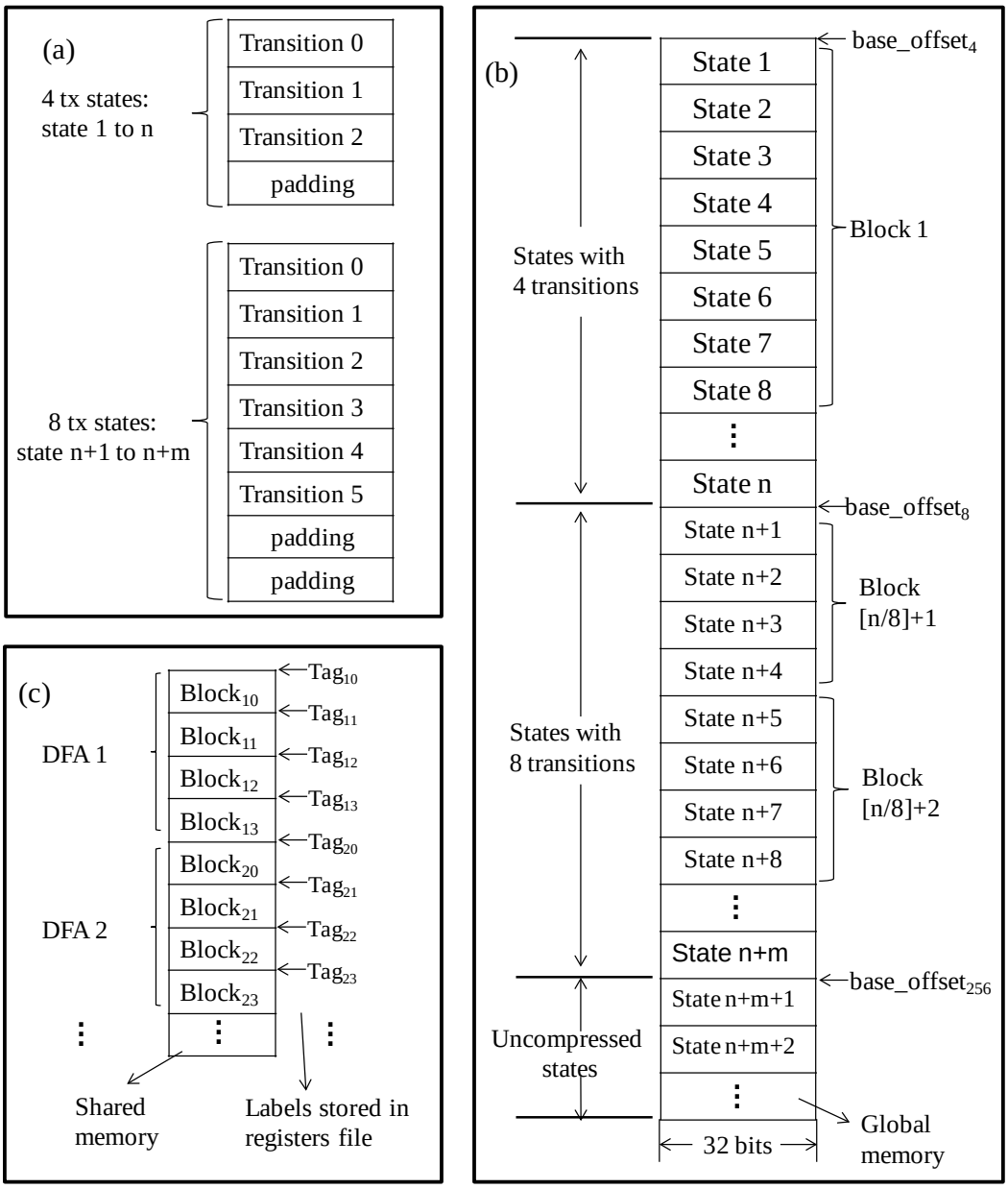


Figure 5: Memory layout of our enhanced compressed DFA solution: (a) layout of compressed states; (b) global memory layout of each DFA; and (c) shared memory layout.

to identify cache hits and misses. We choose to tag each block with the identifier of the first state belonging to it. Since blocks within the same memory region contain an equal number of states, and since the start (and end) of each memory region is marked by the *base_offset* variables, this simple mechanism allows the efficient detection of

cache hits/misses and the identification of the block to be retrieved from global memory in case of a cache miss.

The last two design questions that we consider are: (i) what is the maximum number of cached blocks per DFA, and (ii) where are the tags stored. Our experimental evaluation shows the best results when caching a maximum of 4 blocks per DFA and storing the tags in registers (rather than in shared memory). The size of each cache block is determined dynamically based on the shared memory size and the number of DFAs. The cache design is exemplified in Figure 5(c).

3.4 Hybrid-FA-based Engines

The idea at the basis of the Hybrid-FA proposal [11] is the following: state explosion can be avoided (or limited) by interrupting NFA-to-DFA transformation on states representing repetitions of wildcards and large character sets. This leads to a hybrid automaton with a head-DFA and some tail-NFAs. In turn, the tail-NFAs can be transformed into DFAs (or into hybrid-FA), leading to a hierarchical-DFA solution. The advantage of a mixed DFA-NFA representation of a hybrid-FA is its simplicity, the advantage of a hierarchical-DFA representation is a better bound on the worst-case per character processing. In this work we limit ourselves to hybrid-FAs in the mixed DFA-NFA form. We represent the head-DFA through an uncompressed DFA, and the tail-NFAs through our optimized iNFAnt design. The head- and tail-processing are combined in a single GPU kernel, to ensure that the head- and tail-automata are

processed in lock-step. For more details on hybrid-FAs, the interested reader may refer to [6].

We note that tail-NFAs frequently originate from unbounded repetitions of wildcards (".*") or large character ranges (e.g., "*[^\r\n]**"). Once activated, the former will remain active during the whole traversal, whereas the latter will be deactivated by any character excluded from the repetition (e.g., \r or \n). In our implementation, we introduce a self-checking mechanism that turns off inactive tail-NFAs. We observe that many practical datasets include *[^\r\n]** sub-patterns; in addition, carriage-return and line-feed are frequent characters in textual inputs. As we will show in the experimental evaluation, for these datasets, hybrid-FAs are typically a better option than multiple-DFAs.

CHAPTER 4

EXPERIMENTAL EVALUATION

4.1 Datasets and Platform

We evaluated all our implementations on a system consisting of an Intel Xeon E5620 CPU and an NVIDIA GTX 480 GPU. This device (whose price is currently of about $250) contains 15 streaming multiprocessor, each consisting of 32 cores, and is equipped with about 1.5 GB of global memory.

In our evaluation, we use both real and synthetic pattern-sets. The former have been drawn from Snort's *backdoor* and *spyware* rules (snapshot from December 2011). The synthetic datasets have been generated using the tool described in [34] and using tokens extracted from the backdoor rules. In particular, we generated nine synthetic pattern sets: *exact-match (or EM)*, *range.5*, *range1*, *not-newline.05(nnl.05)*, *not-newline.1(nnl.1)*, *dotstar.5*, *dotstar.1* and *dotstar.2*. As the names suggest, the *exact-match* set contains only exact-match patterns; in *range.5* and *range1* 50% and 100% of the patterns include character sets; in addition to that, the *dotstar** datasets contain a varying fraction of unbounded repetitions of wildcards (5%, 10%, and

20%, respectively). Finally, the *nnl** rules contains a variable fraction of *[^\n\r]** terms, which are very common in Snort. All synthetic datasets consists of 1,000 regular expressions.

The packet traces used in the evaluation have been also synthetically generated using the tool described in [34]. This tool allows producing traces that simulate various amount of malicious activity. This is possible by tuning p_M, a parameter that indicates the probability of malicious traffic. In the generation, we used 15 probabilistic seeds and 4 p_M values: 0.35, 0.55, 0.75, and 0.95. All traces have a 1 MB size. The packet size has been set to 64KB across all experiments. As detailed below, various numbers of packet-flows have been used.

4.2 Dataset Characterization

A characterization of the datasets and of their memory requirements on GPU using different automata representation is reported in Table 1. Because of their simplicity, the *exact-match* and *range** datasets can be compiled into a single DFA, and do not require a hybrid-FA representation. State explosion starts originating when considering pattern-sets with unbounded repetitions of wildcards (i.e., *dotstar** datasets). As can be seen, the number of DFAs generated increase quickly with the fraction of dot-star terms in the pattern-set. The memory footprints are limited in case of NFAs and larger in case of DFAs (even in their default transition compression form). The Hybrid-FA representation represents a compromise between the two. The

Table 1: Dataset characterization.

Dataset	# regex	NFA			DFA							
		# states	# tx	Mem (MB)	# DFA	# states	Uncompressed-DFA		Compressed-DFA		Enhanced-DFA	
							# tx	Mem (MB)	# tx	Mem (MB)	# tx	Mem (MB)
Backdoor	226	4.3k	70.8k	0.54	13	960.1k	245.8M	942	20.27M	81	32M	126
Spyware	462	7.7k	66.8k	0.51	19	680.2k	174.1M	667	6.5M	27.5	21.6M	85
EM	1k	28.7k	51.9k	0.40	1	28.7k	7.35M	28.1	-	-	-	-
Range.5	1k	28.5k	91.8k	0.70	1	41.8k	10.7M	41	-	-	-	-
Range1	1k	29.6k	117.9k	0.90	1	54.4k	13.9M	53.3	-	-	-	-
Dotstar.05	1k	29.1k	116.8k	0.89	13	251k	64.26M	246	1.36M	6.2	2.35M	10
Dotstar.1	1k	29.2k	115.7k	0.88	21	603.8k	154.57M	592	3.37M	15.2	6.2M	26
Dotstar.2	1k	28.7k	114.6k	0.87	32	1.6M	418.1M	1,601	7.26M	33.9	15M	63.4

uncompressed-DFA representation generally has 40-50 times the memory requirements of its compressed counterpart. The *dotstar.2* datasets could not fit the device memory when an uncompressed-DFA representation was used. The enhanced-DFA representation is slightly less memory efficient (by a factor ~1.5X) than the compressed one. The *dotstar.2* datasets cannot fit the device memory when an uncompressed-DFA representation is used.

4.3 Performance Evaluation

4.3.1 Performance comparison of all solutions

A performance evaluation of the proposed solutions on different datasets and packet traces is presented in Figure 6 (a)-(j). The missing data (i.e., bars with value zero) represent unnecessary or unsupported implementations. As mentioned above, simple pattern-sets (*exact-match* and *range**) can be compiled into a single DFA with

limited memory requirements and therefore do not require DFA compression or hybrid-FA solution; on the other hand, complex datasets (*dotstar.2*) do not fit the memory capacity of the GPU (1.5 GB) unless default transition compression is performed. The data show the performance achieved using the optimal number of packet-flows per SM in every implementation. This optimal number varies from case to case, as detailed in Table 2. For completeness, we show also the throughput obtained using the serial CPU implementation described in [8].

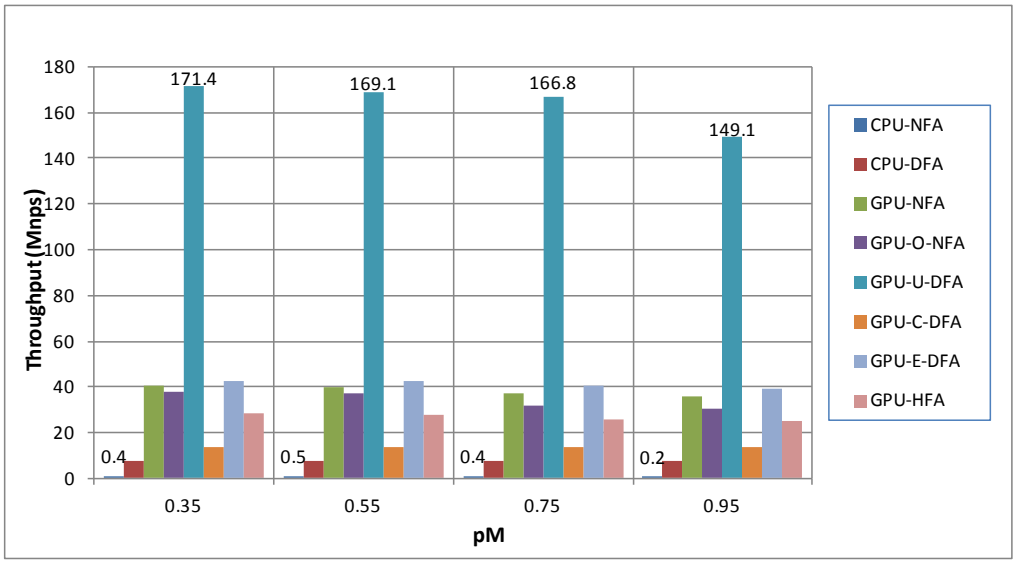


Figure 6 (a): Performance of all implementations on the *backdoor* dataset.

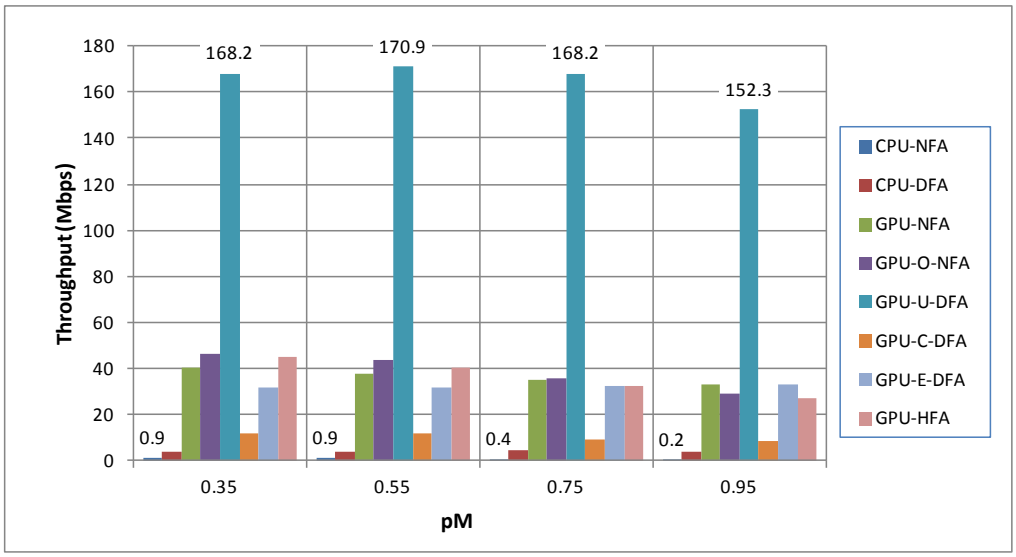


Figure 6 (b): Performance of all implementations on the *spyware* dataset.

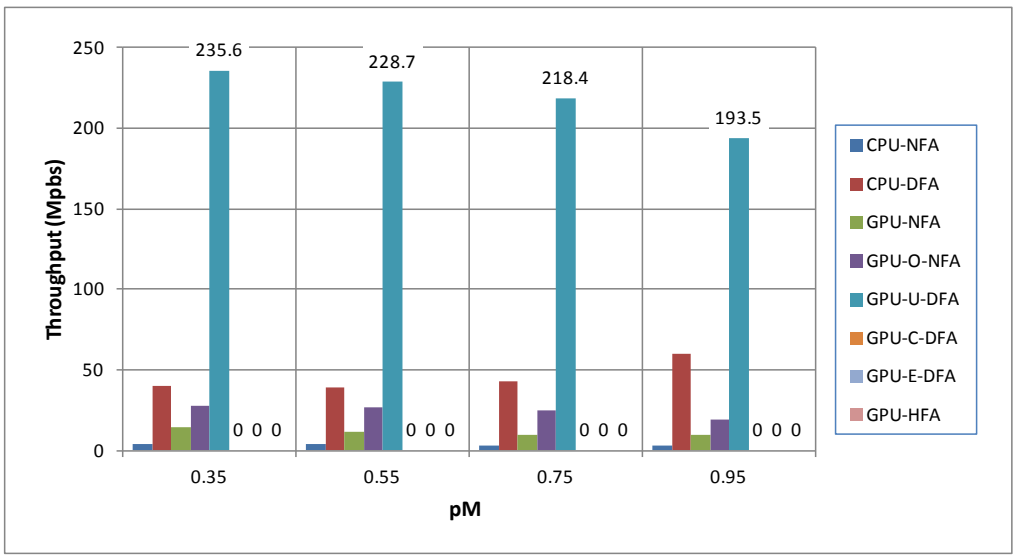


Figure 6 (c): Performance of all implementations on the *exact-match* dataset

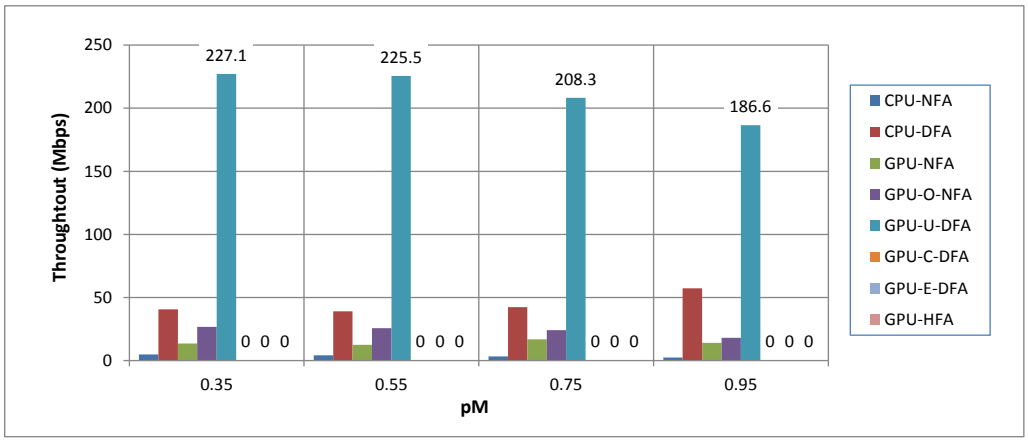


Figure 6 (d): Performance of all implementations on the *range0.5* dataset

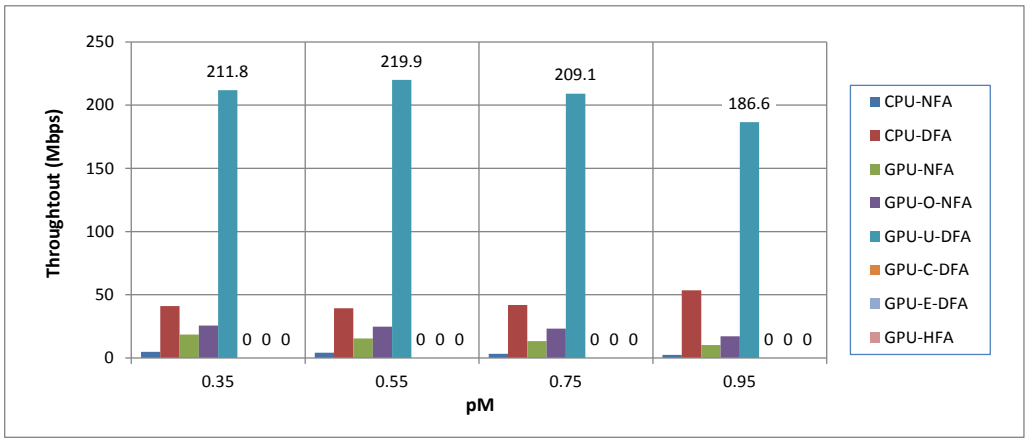


Figure 6 (e): Performance of all implementations on the *range1* dataset

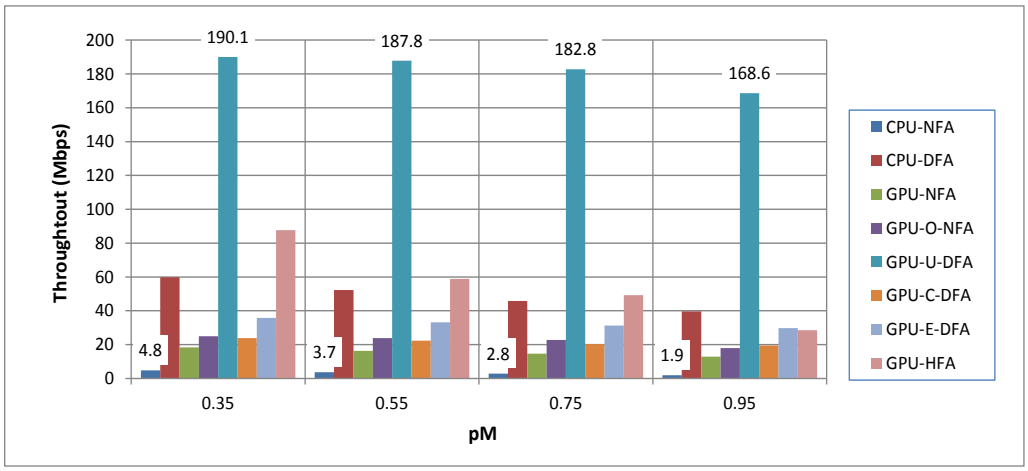


Figure 6 (f): Performance of all implementations on the *nnl0.05* dataset

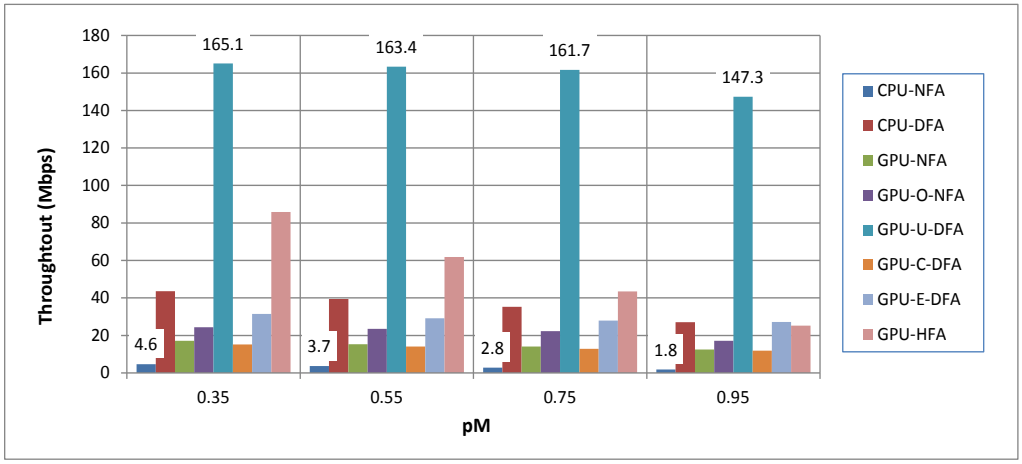


Figure 6 (g): Performance of all implementations on the *nnl0.1* dataset

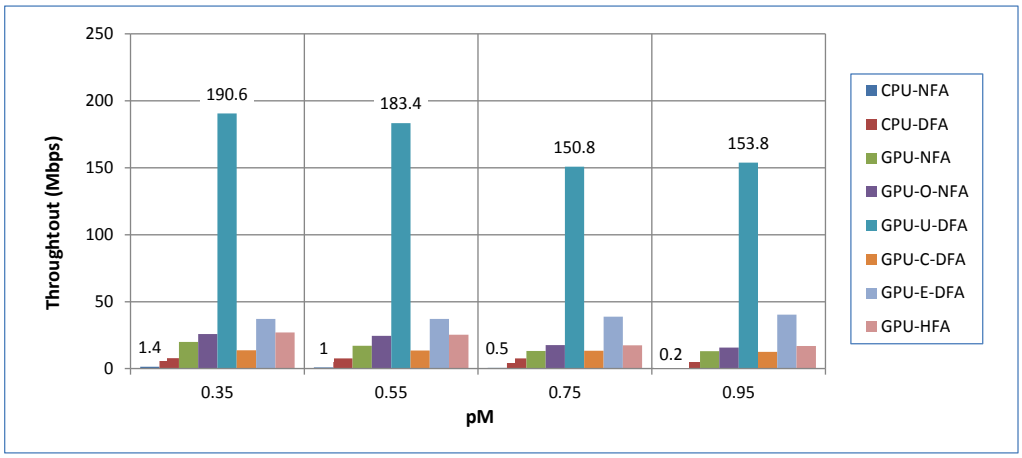


Figure 6 (h): Performance of all implementations on the *dotstar0.05* dataset

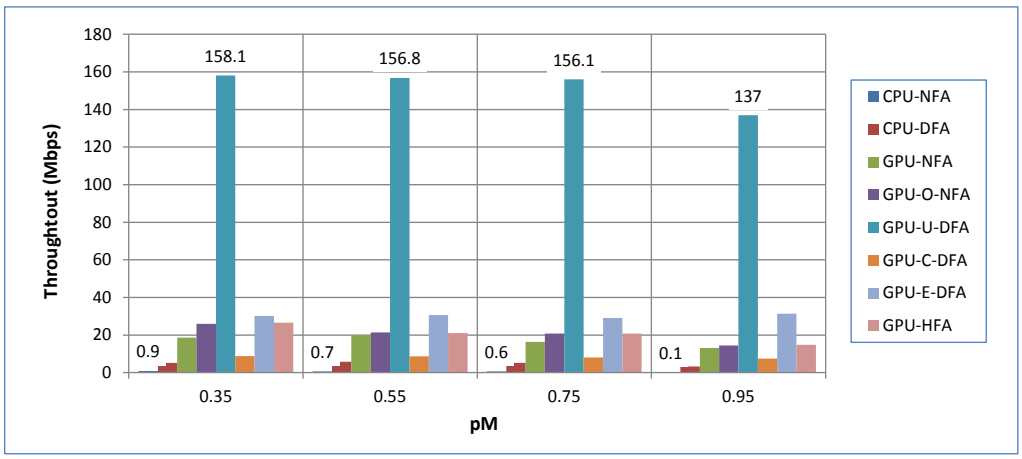


Figure 6 (i): Performance of all implementations on the *dotstar0.1* dataset

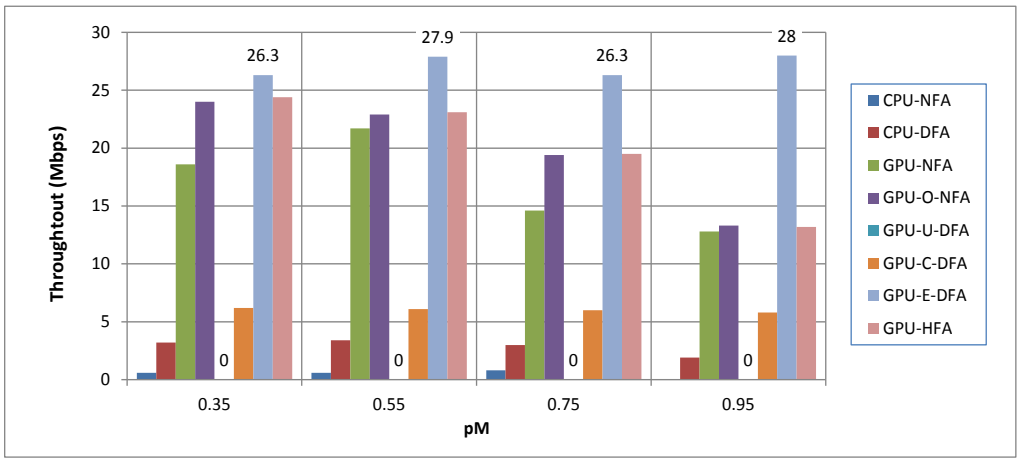


Figure 6 (j): Performance of all implementations on the *dotstar0.2* dataset

We recall that the uncompressed DFA (U-DFA) traverses exactly one state per input character, independent of the pattern-set and trace. The compressed DFA (C-DFA) performs between 1 and 2 state traversals per character due to the presence of non-consuming default transitions: the exact number depends on the characteristics of the underlying pattern-set and of the packet traces. In the enhanced DFA (E-DFA), the majority of the states are compressed and a few are not. The number of state traversals per character in the NFA implementations is also related to the complexity of the patterns and to the nature of the input stream. In our experiments, the average number of state traversals per character using an NFA scheme varied from 1.8 (on the *exact-match* and *range** datasets and p_M=0.35) to 190 (on the *dotstar.2* dataset and p_M=0.95). For hybrid-FAs, the size of the active set is mostly contained (from 1.1 to 17.6) for low p_M (0.35 and 0.55), but approaches that of NFAs for p_M=0.95. The number of state traversals per character ultimately affects the performance.

From Figure 6, we can see that U-DFA, whenever applicable, is the best solution across all pattern-sets. This is due to the simplicity and regularity of its computation. However, because of its high memory requirements, this solution is not applicable to complex pattern-sets including a fraction of rules with wildcard repetitions ≥ 2% (for example, the *dotstar.2* dataset). E-DFA is a good compromise between U-DFA and C-DFA: it achieves a 3-5X speedup over C-DFA at the cost of ~1.5X its memory requirements. On synthetic datasets, the performance improvement increases with the complexity of the patterns (i.e., with the fraction of wildcard repetitions). As explained in Section 3.3.3., this performance gain is due to the more regular computation and better thread utilization of E-DFA over C-DFA. Further, E-DFA outperforms both NFA solutions on almost all datasets, and is more resilient to malicious traffic patterns. All DFA-based GPU implementations greatly outperform their CPU counterparts.

Our optimized NFA implementation (O-NFA) achieves a speedup over iNFAnt (NFA) by reducing the number of transitions processed on every input character. In our experiments, the number of iterations over the loop at line 6 in the iNFAnt pseudo-code is reduced up to a 5X factor. This reduction leads to a performance improvement, which, however, is not so dramatic. This is because the number of iterations is not the only factor that contributes to the matching speed. The additional atomic operation and the more complex control-flow are limiting factors to the speedup. Both NFA-based GPU implementations outperform their CPU counterpart

by a factor varying from ~10X (for simple patterns and low p_M) to ~80X (for complex patterns and high p_M).

The performance of the Hybrid-FA implementation varies greatly across pattern-sets and input traces. Hybrid-FAs outperform compressed DFAs for low p_M, since in these cases the traversal is mostly limited to the head-DFA. However, such representation is penalized by the presence of malicious traffic (i.e., high values of p_M), that trigger the activation of a number of tail-states. The *nnl** datasets, however, exhibit better characteristics. This is due to the fact that their tail-NFAs are mostly deactivated every time a new line or a carriage return character is processed. This happens pretty frequently on input traces containing textual information.

4.3.2 Evaluation of the multiple flows' processing

The optimal number of packet-flows per SM varies across the implementations, pattern-sets and traces. Our results are summarized in Table 3. Most implementations reach their peak performance at 4-5 flows/SM. However, in the case of complex datasets, C-DFA achieves best performances at 2-3 flows/SM. Recall that, in C-DFA, bi-dimensional thread-blocks are used. The block-size is equal to 32 along the *x*-dimension, and is equal to the number of DFAs along the *y*-dimension. In case of complex datasets, the large number of DFAs leads to large thread-blocks that fully utilize the SM. Therefore, further performance improvements cannot be achieved by increasing the flow-level parallelism beyond 2-3 flows/SM.

Table 2: Effect of number of flows/SM on performance.

Implementation	Optimal # flows per SM	Improvement over 1 flow per SM	
		Min	Max
U-DFA	5	1.72	2.75
C-DFA	5 (single-DFA) 3 (multi-DFAs)	1.16	2.82
E-DFA	5	2.49	3.33
iNFAnt	4	1.81	3.50
Opt-iNFAnt	4	2.55	3.65

4.3.3 Evaluation of the proposed caching scheme

All data presented so far have been reported by allowing the GPU to automatically treat part of the shared memory as a hardware-managed cache. We now evaluate the effect of our software-managed cache design on the performance of E-DFA. Figure 7 (a)-(e) show the cache miss rates of all datasets. We tested various numbers of cached blocks per DFA—2,4, and 8, and set the cache block size to the multiple of 128B which allows the maximum utilization of the shared memory (such size depends on the number of DFAs processed in parallel). Intuitively, more blocks will lead to lower miss rate. From Figure 7 (a)-(e) we can see that: (i) 2 cached blocks per DFA result in the highest miss rate on all datasets; (ii) 4 and 8 blocks per DFA lead to similar miss rates; (iii) in some cases, 4 blocks per DFA is the best configuration. We want to find a good balance between number of blocks and blocks size. Figure 8 (a)-(e) show the throughput achieved when processing a single flow per SM using different cache configurations (no cache, and software-managed cache with

2, 4, and 8 blocks per DFA). In case of caching, the tag information is stored in registers. As can be seen, the use of caching generally allows better performance, especially when using 4-8 blocks per DFA. Using 4 blocks leads to similar or even better performance than using 8 blocks: the overhead due to managing more blocks counteracts the slight advantage in terms of miss rate. We conclude that, in case of one flow per SM, software-based caching with 4 blocks per DFA is the best configuration. It must be noted that the performance gain depends on the characteristics of the input traffic: average traffic (p_M=0.35 and p_M=0.55) exhibits good locality behavior and therefore lower miss rate, leading to better performance. However, malicious traffic (p_M=0.55 and p_M=0.75) exhibits worse locality behavior and higher miss rate, and, as a consequence, reports little-to-no performance gain in the presence of caching. The complexity of the dataset also affects the performance: a higher number of DFAs leads to smaller blocks and consequently to higher miss rate.

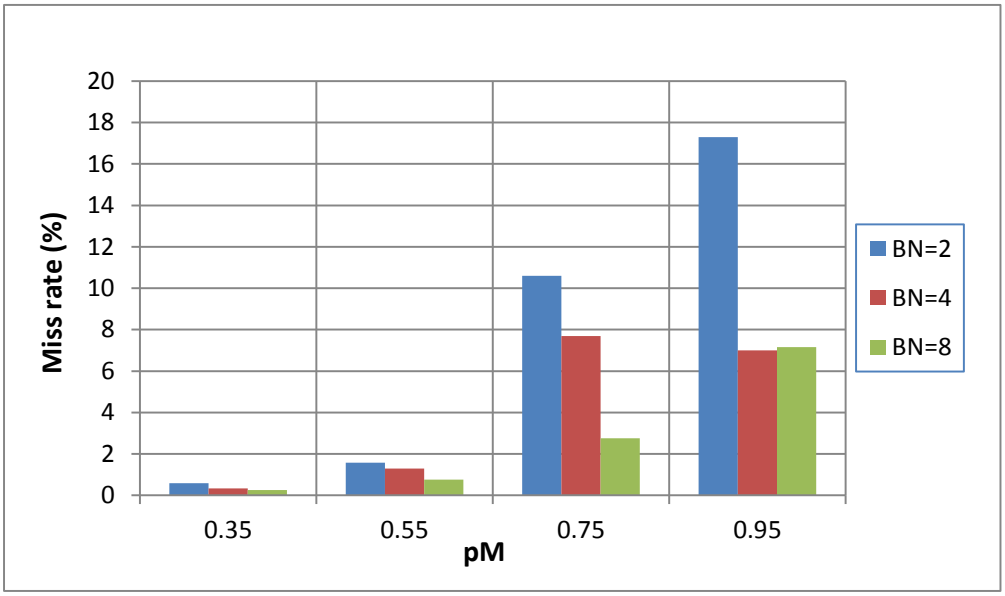


Figure 7 (a): Miss rate of *backdoor* dataset for various numbers of blocks (BN) and probability of malicious traffic (p_M).

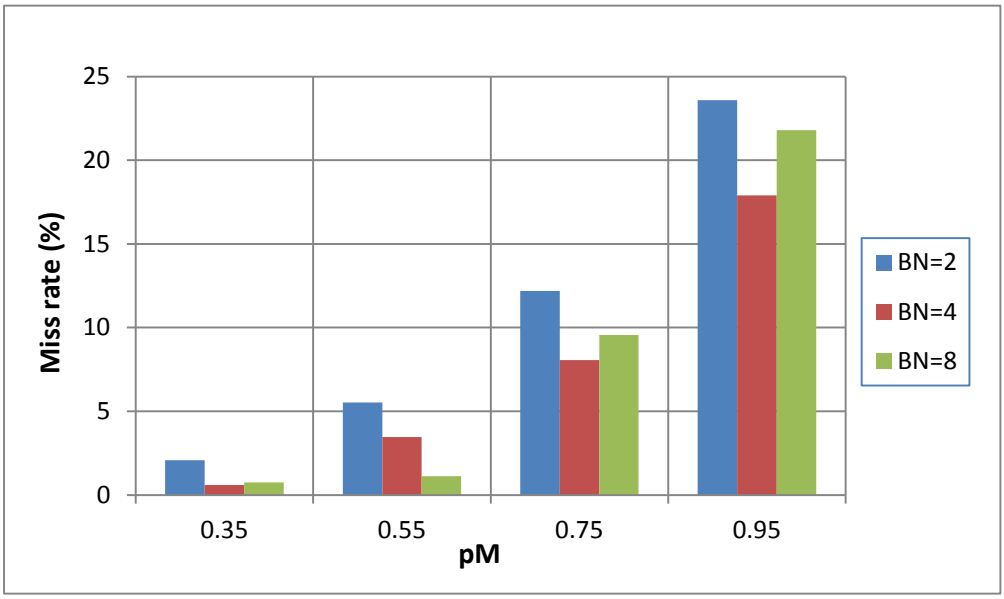


Figure 7 (b): Miss rate of *spyware* dataset for various numbers of blocks (BN) and probability of malicious traffic (p_M).

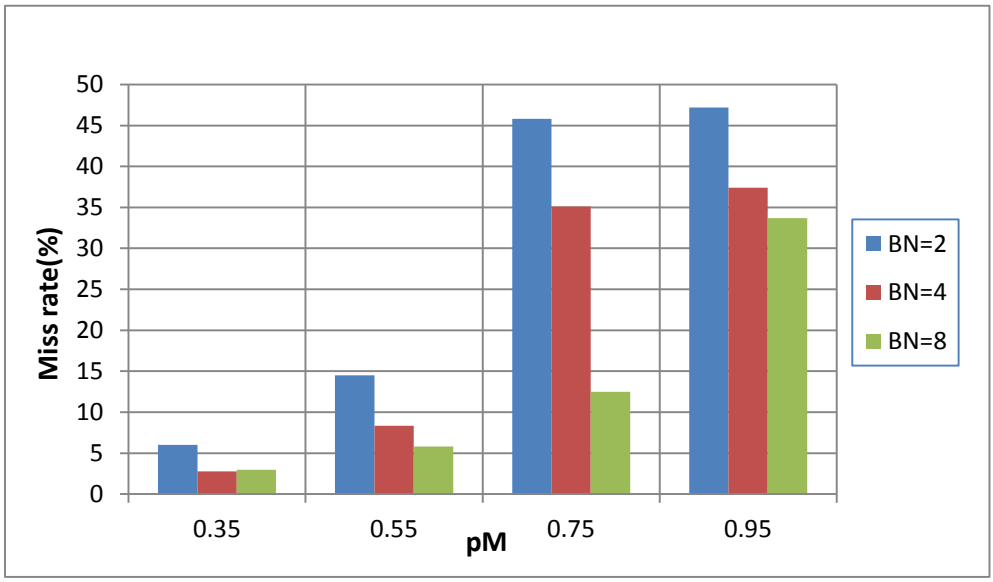


Figure 7 (c): Miss rate of *dotstar0.05* dataset for various numbers of blocks (BN) and probability of malicious traffic (p_M).

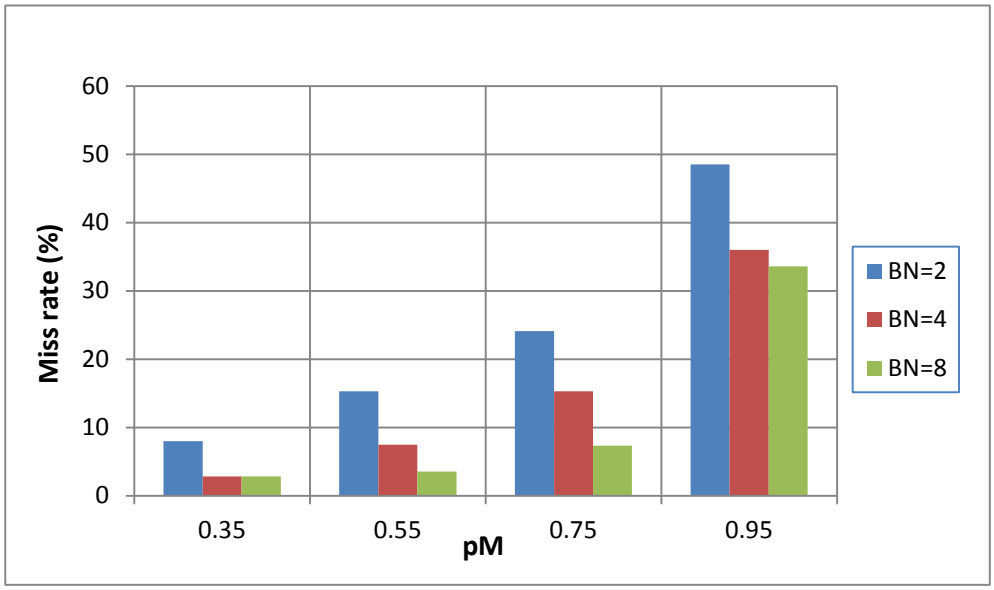


Figure 7 (d): Miss rate of *dotstar0.1* dataset for various numbers of blocks (BN) and probability of malicious traffic (p_M).

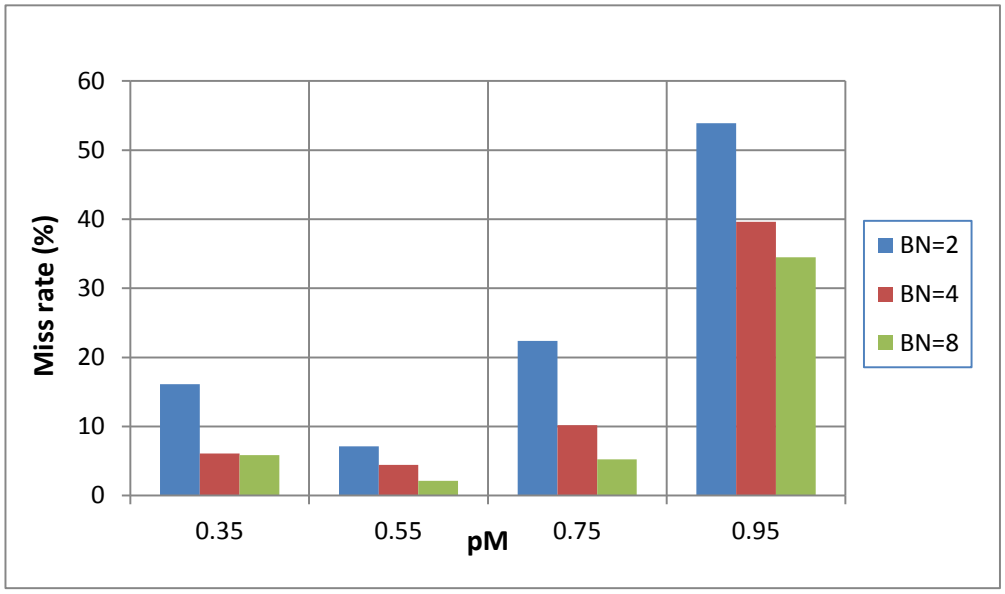


Figure 7 (e): Miss rates of *dotstar0.2* dataset for various numbers of blocks (BN) and probability of malicious traffic (p_M).

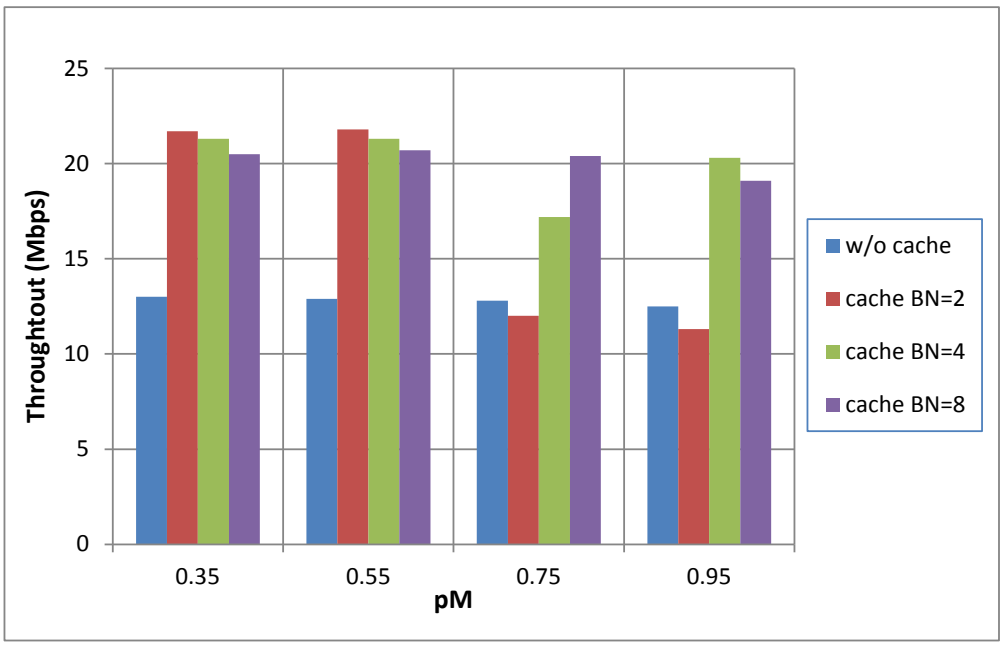


Figure 8 (a): Performance of *backdoor* dataset for various numbers of blocks (BN) and probability of malicious traffic (p_M).

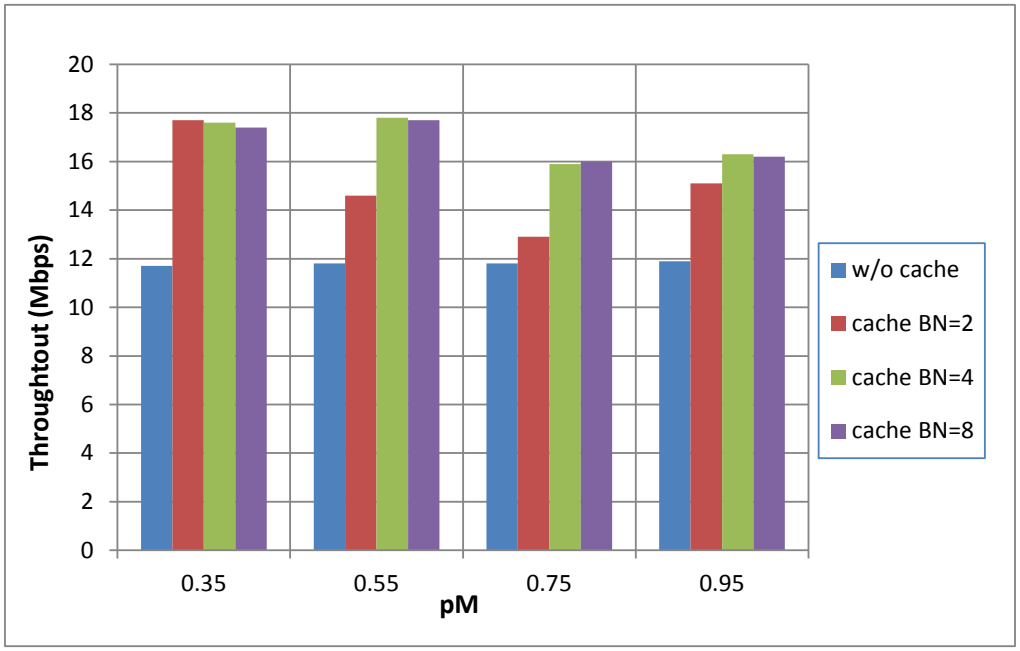


Figure 8 (b): Performance of *spyware* dataset for various numbers of blocks (BN) and probability of malicious traffic (p_M).

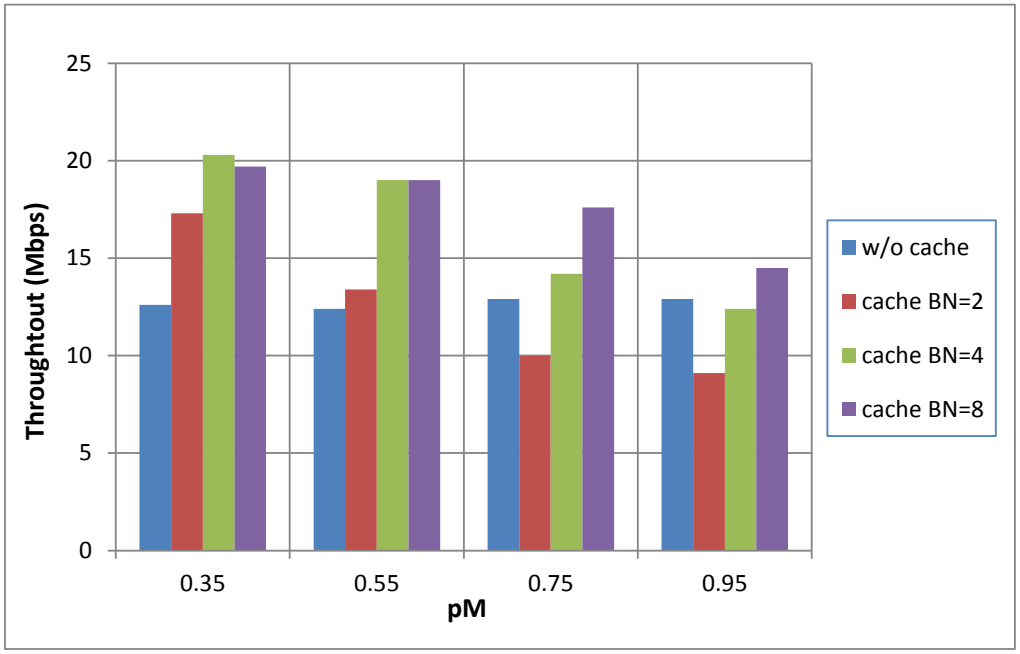


Figure 8 (c): Performance of *dotstar0.05* dataset for various numbers of blocks (BN) and probability of malicious traffic (p_M).

Finally, we verified that using a software-managed cache does not improve the performance when increasing the number of flows mapped onto the same SM. Recall that each flow is associated to a thread-block. Since, with the described caching scheme, each thread-block fully utilizes the share memory, the execution of multiple thread-blocks mapped onto the same SM is serialized by the warp scheduler. To allow concurrent execution of flows mapped onto the same SM it is necessary to reduce their shared memory requirement. This can be done by using small cache blocks (i.e., 128-byte blocks independently of the number of DFAs). Figure 9 (a)-(e) show the performance comparison of no software-managed caching, caching with multi-128B blocks, and caching with 128B blocks on all considered datasets. We tested these configurations on the 1 flow per SM and 5 flows per SM cases. In the 1 flow per SM

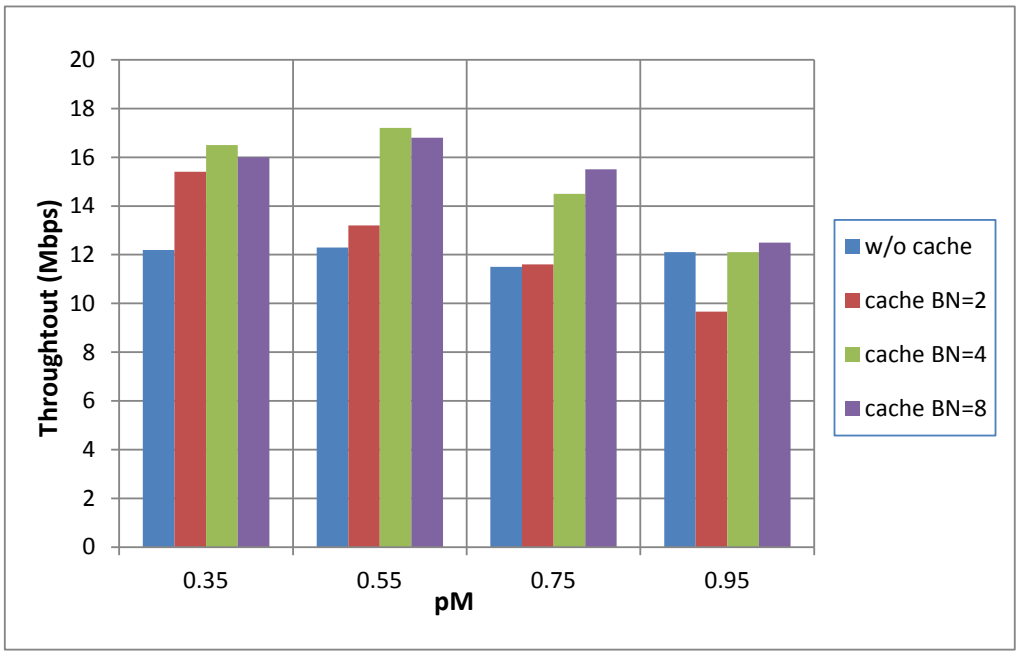


Figure 8 (d): Performance of *dotstar0.1* dataset for various numbers of blocks (BN) and probability of malicious traffic (p_M).

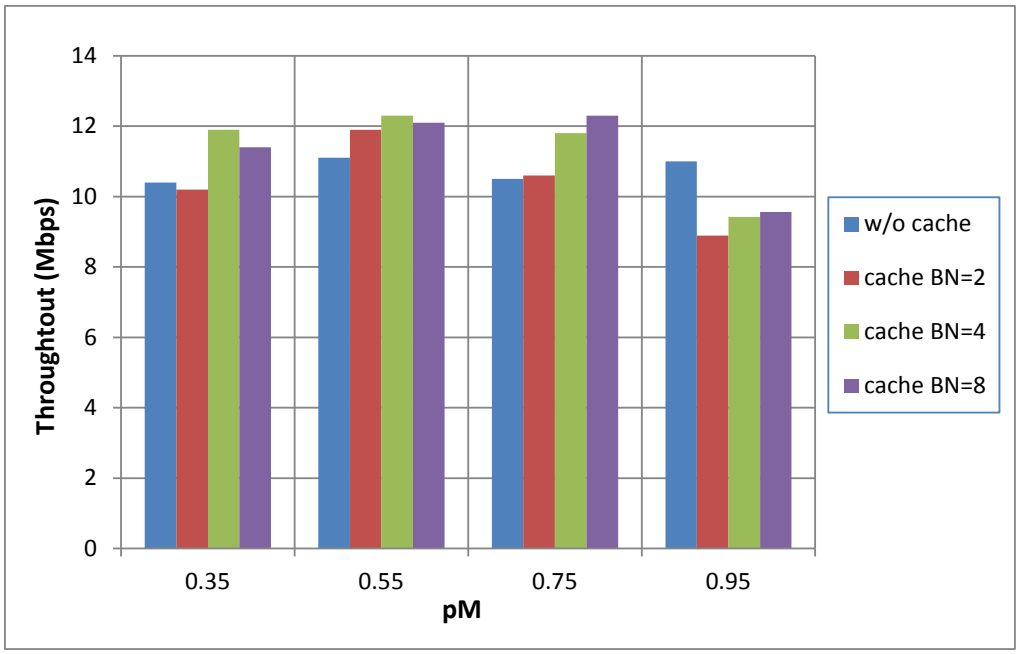


Figure 8 (e): Performance of *dotstar0.2* dataset for various numbers of blocks (BN) and probability of malicious traffic (p_M).

case, the use of small (128B) cache blocks leads to higher miss rates, and thus to negligible performance gains. On the other hand, in the 5 flows per SM case the use of large (multi 128B blocks) leads to little improvement; better performance is achieved with small blocks or no caching. In summary, even if regular expression matching exhibits good locality, the limited size of the shared memory does not make the use of ad-hoc caching schemes particularly advantageous on GPUs. Higher performance gains can be achieved by exploiting flow-level parallelism rather than by making use of caching.

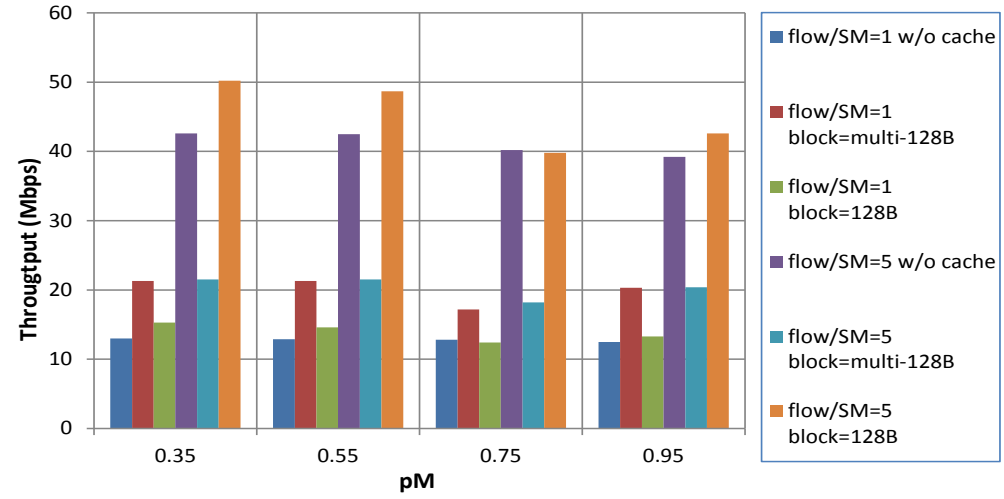


Figure 9 (a): *Backdoor* dataset: performance of different cache configurations with single- and multi-flow processing.

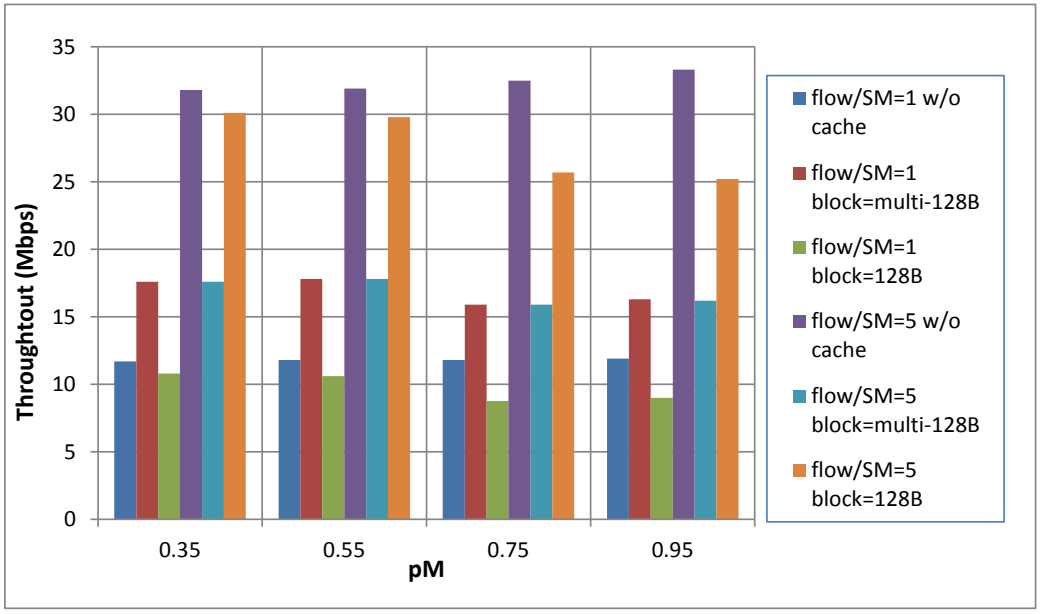


Figure 9 (b): *Spyware* dataset: performance of different cache configurations with single- and multi-flow processing.

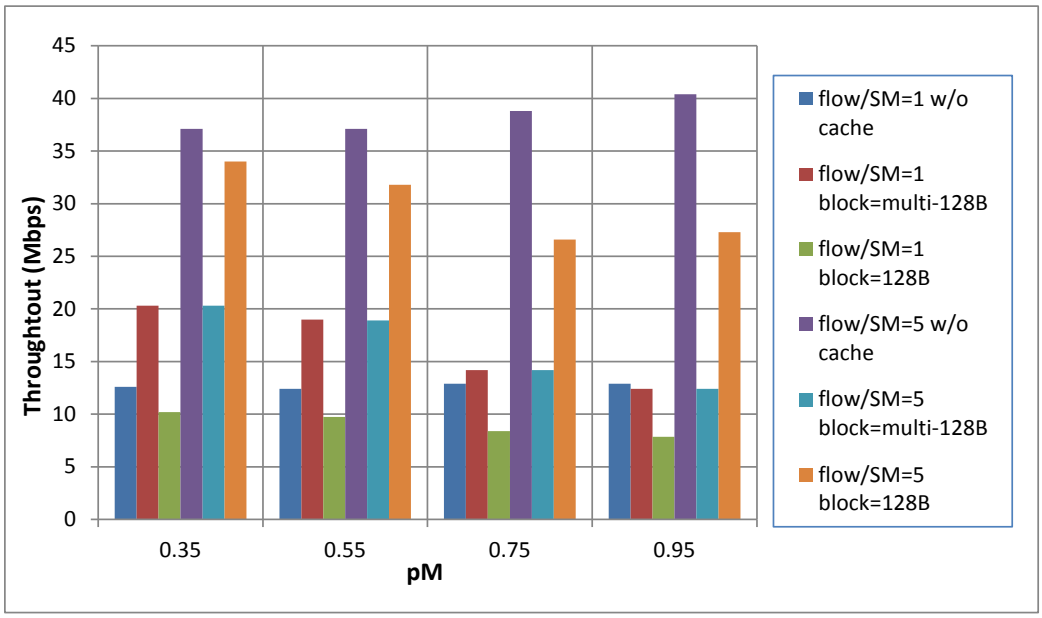


Figure 9 (c): *Dotstar0.05* dataset: performance of different cache configurations with single- and multi-flow processing.

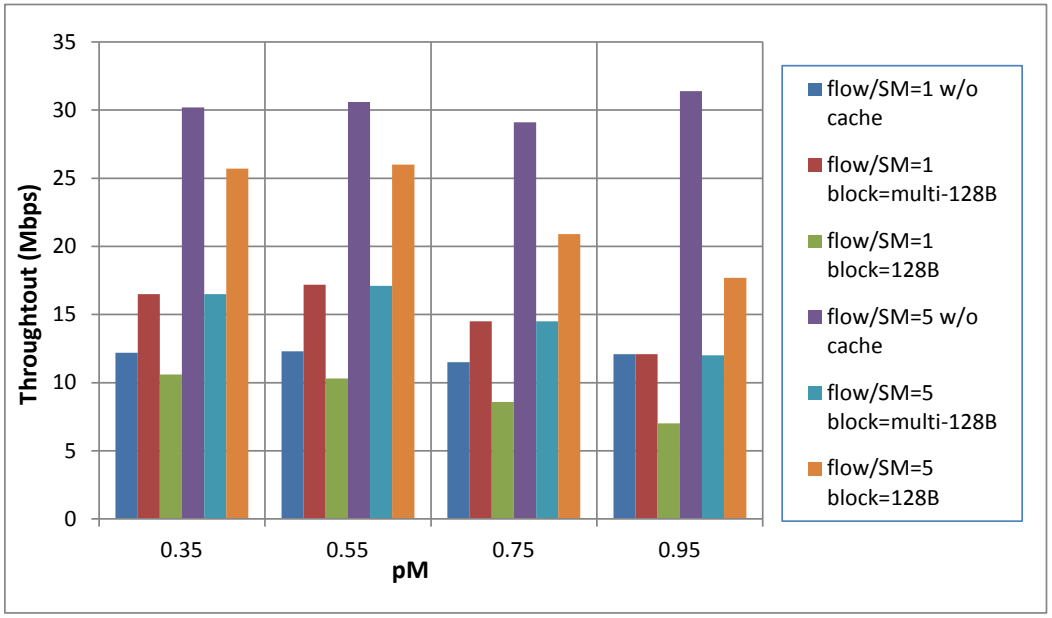


Figure 9 (d): *Dotstar0.1* dataset: performance of different cache configurations with single- and multi-flow processing.

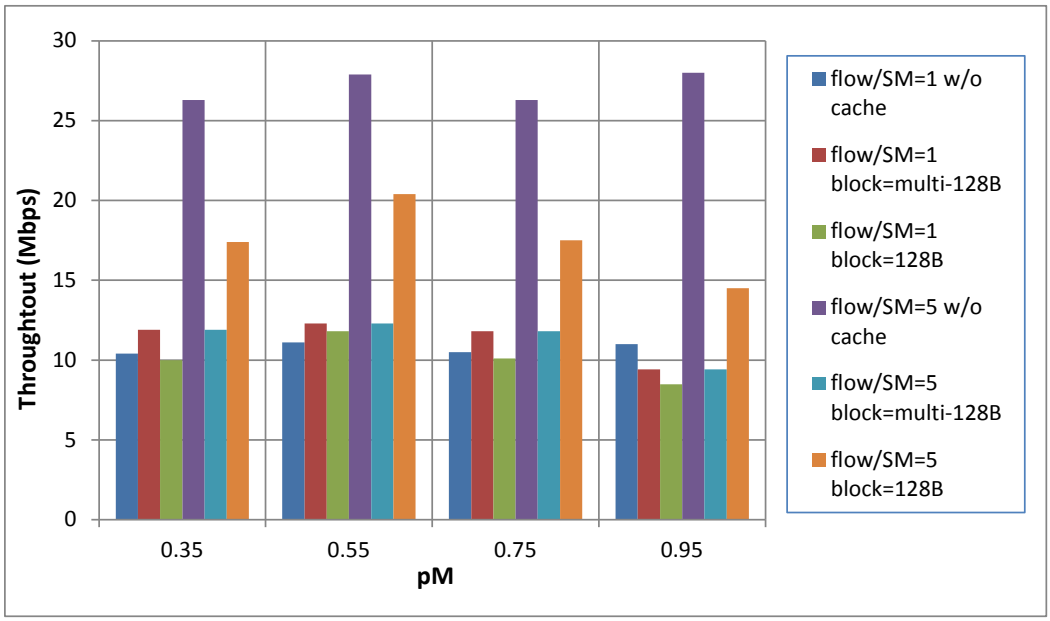


Figure 9 (e): *Dotstar0.2* dataset: performance of different cache configurations with single- and multi-flow processing.

CHAPTER 5

CONCLUSION

In this work, we have provided a comprehensive study of regular expression matching on GPUs. To this end, we have used datasets of practical size and complexity and explored advantages and limitations of different NFA- and DFA-based representations. We have taken advantage of the hardware features of GPU in order to provide efficient implementations.

Our evaluation shows that, because of the regularity of its computation, an uncompressed DFA solution outperforms other implementations 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. We have shown schemes to improve a basic default-transition compressed DFA design so to allow more regular processing and better thread utilization. We have also shown that, because of the limited on-chip memory available on GPU, the use of elaborate caching scheme does not allow substantial performance improvements on large number of packet flows.

REFERENCES

[1] J. Newsome, B. Karp, and D. Song, “Polygraph: automatically generating signatures for polymorphic worms,” in Symp. Security & Privacy 2005, pp. 226-241.

[2] R. Sommer, and V. Paxson, “Enhancing byte-level network intrusion detection signatures with context,” in Proc. of CCS 2003, pp. 262-271.

[3] Y. Xie et al., “Spamming botnets: signatures and characteristics,” in Proc. of ACM SIGCOMM 2008, pp. 171-182.

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

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

[6] M. Becchi, and P. Crowley, “A hybrid finite automaton for practical deep packet inspection,” in Proc. of CoNEXT 2007.

[7] M. Becchi, and P. Crowley, “Efficient regular expression evaluation: theory to practice,” in Proc. of ANCS 2008, pp. 50-59.

[8] M. Becchi, C. Wiseman, and P. Crowley, “Evaluating regular expression matching engines on network and general purpose processors,” in Proc. of ANCS 2009, pp. 30-39.

[9] S. Kumar et al., “Curing regular expressions matching algorithms from insomnia, amnesia, and acalculia,” in Proc. of ANCS 2007.

[10] S. Kumar et al., “Algorithms to accelerate multiple regular expressions matching for deep packet inspection,” in Proc. of ICNP 2006, pp. 339-350.

[11] S. Kumar, J. Turner, and J. Williams, “Advanced algorithms for fast and scalable deep packet inspection,” in Proc. of ANCS 2006.

[12] F. Yu et al., “Fast and memory-efficient regular expression matching for deep packet inspection,” in Proc. of ANCS 2006.

[13] B. C. Brodie, D. E. Taylor, and R. K. Cytron, “A Scalable Architecture For High-Throughput Regular-Expression Pattern Matching,” in Proc. of ISCA 2006, pp. 191-202.

[14] R. Smith et al., “Deflating the big bang: fast and scalable deep packet inspection with extended finite automata,” in Proc. of SIGCOMM 2008, pp. 207-218.

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

[16] R. Sidhu, and V. K. Prasanna, “Fast Regular Expression Matching Using FPGAs,” in Proc. of FCCM 2001, pp. 227-238.

[17] C. R. Clark, and D. E. Schimmel, “Efficient Reconfigurable Logic Circuits for Matching Complex Network Intrusion Detection Patterns,” in Proc. of FPL 2003.

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

[19] G. Vasiliadis et al., “Gnort: High Performance Network Intrusion Detection Using Graphics Processors,” in Proc. of RAID 2008.

[20] G. Vasiliadis et al., “Regular Expression Matching on Graphics Hardware for Intrusion Detection,” in Proc. of RAID 2009.

[21] R. Smith et al., “Evaluating GPUs for network packet signature matching,” in Proc. of ISPASS 2009, pp. 175-184.

[22] Niccolo’Cascarano et al., “iNFAnt: NFA Pattern Matching on GPGPU Devices,” ACM SIGCOMM Computer Communication Review, vol. 40 Num. 5, pp. 21-26, 2010.

[23] Y. Zu et al., “GPU-based NFA implementation for memory efficient high speed regular expression matching,” in Proc. of PPOPP 2012, pp. 129-140.

[24] M. Becchi, and P. Crowley, “Extending finite automata to efficiently match Perl-compatible regular expressions,” in Proc. of CoNEXT 2008, pp. 1-12.

[25] C. R. Meiners et al., “Fast regular expression matching using small TCAMs for network intrusion detection and prevention systems,” in Proc. of USENIX Conference on Security, 2010.

[26] C. R. Meiners, A. X. Liu, and E. Torng, “Bit Weaving: A Non-Prefix Approach to Compressing Packet Classifiers in TCAMs,” in TON, vol. 20, no. 2, pp. 488-500, 2012.

[27] K. Peng et al., “Chain-Based DFA Deflation for Fast and Scalable Regular Expression Matching Using TCAM,” in Proc. of ANCS 2011, pp. 24-35.

[28] S. Kong, R. Smith, and C. Estan, “Efficient signature matching with multiple alphabet compression tables,” in Proc. of Securecomm 2008, pp. 1-10.

[29] D. Tarditi, S. Puri, and J. Oglesby, “Accelerator: using data parallelism to program GPUs for general-purpose uses,” in Proc. of ASPLOS 2006, pp. 325-335.

[30] S. Che et al., “Rodinia: A benchmark suite for heterogeneous computing,” in Proc. of IISWC 2009, pp. 44-54.

[31] V. W. Lee et al., “Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU,” in Proc. of ISCA 2010, pp. 451-460.

[32] J. Nickolls et al., “Scalable Parallel Programming with CUDA,” Queue, vol. 6, no. 2, pp. 40-53, 2008.

[33] S. Han et al., “PacketShader: a GPU-accelerated software router,” in Proc. of SIGCOMM 2010, pp. 195-206.

[34] M. Becchi, M. Franklin, and P. Crowley, “A workload for evaluating deep packet inspection architectures,” in Proc. of IISWC 2008, pp. 79-89.

VITA

Xiaodong Yu

Research Interests

- Data structures and algorithm design for application acceleration on parallel computer architecture;
- High-performance and parallel computer architectures, GPUs, FPGAs, compiler and runtime support;
- High-performance computing and embedded computing;
- Networking systems architectures and implementation, network security, distributed systems;
- Bioinformatics

Education

M.S. Electrical Engineering, University of Missouri, MO, July 2013

B.S. Mathematics and Applied Mathematics, China University of Mining and Technology (CUMT), Xuzhou, China, June 2008

PAPERS AND POSTERS

Conference Paper:

- **Xiaodong Yu** and Michela Becchi, "GPU Acceleration of Regular Expression Matching for Large Datasets: Exploring the Implementation Space," In Proc. of the 10th ACM International Conference on Computing Frontiers (CF 2013), Ischia, Italy, May 2013

Posters:

- **Xiaodong Yu** and Michela Becchi, "Accelerating Regular Expression Matching on Graphics Processing Units," Missouri Informatics Symposium 2012
- **Xiaodong Yu** and Michela Becchi, "Exploring Different Automata Representations for Efficient Regular Expression Matching on GPU," In Proc. of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP 2013), Shenzhen, China, February 2013

Conference Participations

- ACM/IEEE Symposium on Architectures for Networking and Communications

Systems (ANCS 2011), October 3-4, 2011, Brooklyn, NY

- Missouri Informatics Symposium (MIS 2012), October 22-23, 2012, Columbia, MO
- 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2013), Feb. 23-27, 2013, Shenzhen, China

Skills

Programming languages Adept with C, C++, Assembly Language, VHDL, JAVA, PERL, Visual BASIC, SQL

API Packages CUDA, OpenMP, Pthread

Tools MATLAB, VHDL simulation tools, Mathematica, ANSYS, Dreamweaver

Languages Proficient with spoken and written English, Native speaker of Mandarin Chinese