iNFAnt: NFA pattern matching on GPGPU devices

iNFAnt: NFA Pattern Matching on GPGPU Devices

Niccolo' Cascarano, Pierluigi Rolando, Fulvio Risso, Riccardo Sisto
Politecnico di Torino
Turin, Italy
{niccolo.cascarano, pierluigi.rolando, fulvio.risso, riccardo.sisto}@polito.it

ABSTRACT

This paper presents iNFAnt, a parallel engine for regular expression pattern matching. In contrast with traditional approaches, iNFAnt adopts non-deterministic automata, allowing the compilation of very large and complex rule sets that are otherwise hard to treat.

iNFAnt is explicitly designed and developed to run on graphical processing units that provide large amounts of concurrent threads; this parallelism is exploited to handle the non-determinism of the model and to process multiple packets at once, thus achieving high performance levels.

Categories and Subject Descriptors

C.2.3 [**Computer-Communication Networks**]: Network Operations

General Terms

Experimentation, Algorithms

Keywords

NFA, pattern matching, CUDA, GPGPU, regular expression

1. INTRODUCTION

Pattern matching, i.e. the task of matching a string of symbols against a set of given patterns, plays an important role in multiple fields that range from bioinformatics (e.g. for analyzing DNA sequences) to high-speed packet processing, where it is a critical component for packet filtering, traffic classification and, in general, deep packet inspection.

Pattern matching is commonly performed by expressing patterns as sets of regular expressions and converting them into finite state automata (FSAs), mathematical models that represent (potentially infinite) sets of strings. The behavior of FSAs is simple to emulate on computing devices in order to perform the actual matching procedure and finite automata can be easily composed together with the full set of boolean operators.

Two kinds of FSAs are known from the literature, deterministic (DFA) and non-deterministic (NFA). While automata theory proves them equivalent in terms of expressiveness, their practical properties are different: NFA traversal requires, by definition, non-deterministic choices that are hard to emulate on actual, deterministic processors; on the other hand DFAs, while fast to execute, can be less space-efficient, requiring very large amounts of memory to store certain peculiar patterns that are rather common in practice (the so-called *state space explosion*) [2]. In general, software-based NFA implementations suffer from a higher per-byte traversal cost when compared with DFAs: intuitively, this is because multiple NFA states can be active at any given step, while only a single one must be considered when processing DFAs.

So far software research has been focused mainly on DFAs, as they provide a relatively easy way to achieve high throughputs; many efforts have been aimed at solving the inherent downsides of this model and avoid the aforementioned memory explosion. At the same time, NFAs have often been relegated to the design of hardware devices (e.g. FPGAs) that can easily mimic their behavior, or for use where high throughput is not the primary concern (e.g. many general-purpose pattern-matching libraries).

This paper presents iNFAnt, a NFA-based regular expression engine running on graphical processing units (GPUs). iNFAnt represents a significant departure from traditional software-based pattern matching engines both for its underlying automaton model, the NFA, and its target hardware platform, the GPU. NFA adoption allows iNFAnt to efficiently store very large regular expression sets in a limited amount of memory while the parallelism offered by the underlying hardware helps countering the higher per-byte traversal cost of NFAs with respect to DFAs and the higher instruction execution time of GPUs with respect to CPUs. iNFAnt also represents, as far as we know, one of the first approaches to pattern matching designed from the ground up for the heavily parallel execution environment offered by the modern programmable GPUs, as opposed to being an adaptation of a technique originally designed for general-purpose processors.

2. RELATED WORKS

It is common knowledge that pattern matching is the most time-expensive operation to be performed in intrusion detection systems and similar applications: accelerating its execution has been the object of several academic works. Some of them have already considered the idea of using the parallelism offered by GPUs.

It is possible either to try to execute the whole packet processing application on a graphical device or to accelerate only the pattern matching portion. The first case saw the development of Gnort [7], a full port of the Snort IDS[1] to a GPU environment.

[1]Available at `http://www.snort.org/`.

Gnort did not initially support regular expressions, delegating them to the host CPU; it has been since extended with a DFA-based regex engine [8]. DFA approaches, however, incur in state space explosion, typically solved by heuristically splitting the rules into smaller subsets or (as Gnort does) translating only a "well-behaved" subset while keeping the rest in NFA form for host processing [8]. Both solutions are suboptimal for our goals as splitting leads to inefficiencies (all the DFAs must be traversed) while resorting to host processing defies the goal of graphical hardware adoption. Other more advanced approaches for countering the DFA state explosion problem have been proposed, such as HFAs [1] that split the DFA under construction at the point where state space explosion would happen.

There have been some experiences in porting advanced techniques, such as XFAs [5], to GPUs; however adapting a traversal algorithm designed for CPUs on a GPU-based device is not straightforward or efficient because of deep architectural differences.

Other techniques described in the literature use GPUs to perform preprocessing steps or, alternatively, employ inexact algorithms to perform matching (e.g. [6]). These approaches are out of scope for the purpose of this paper, aimed at a full-fledged regex engine.

Perhaps the work most closely related to iNFAnt is reported in [4] and describes methods to run DFAs and NFAs on a high-speed single-instruction, multiple data (SIMD) processor. While NFAs are recognized as a viable technique on parallel hardware and for reducing memory consumption, the proposed algorithm implements only a subset of the regular expression operators; moreover it considers an architecture radically different from GPUs in terms of specifications and programming model.

3. CUDA ARCHITECTURE

The latest trends have seen a shift towards the development of inexpensive, highly-parallel and programmable GPUs. Employing these processors in fields unrelated to computer graphics has been dubbed *general-purpose computation on graphical processing units* or GPGPU.

There are multiple kinds of programmable GPUs available on the market and only recently a standard programming interface, OpenCL[2], is emerging.

For the purpose of this work, we have used nVidia devices that implement and expose the Compute Unified Device Architecture[3] (CUDA) programming interface.

3.1 Execution cores

CUDA devices are logically composed of arrays of single instruction, multiple threads (SIMT) processors, the *multiprocessors*, each one containing a number of physical execution cores (typically 8). The devices support thousands of threads at the same time, multiplexed on their far smaller set of cores by a dedicated hardware scheduler that avoids the overhead usually associated to context switching. The instruction set is RISC and most instructions require multiple clock cycles for their execution instruction: efficiency comes from the large number of cores, not their individual performance which is low.

[2]Described at `http://www.khronos.org/opencl/`.

[3]Available at `http://www.nvidia.com/object/cuda_home_new.html`.

In the SIMT paradigm each multiprocessor executes the same instruction simultaneously for multiple threads by assigning a different execution context to each core; when this is not possible (e.g. due to the different outcomes of a conditional branch) threads are said to *diverge* and the execution of groups of threads that go along different code paths is sequentialized by the scheduler.

CUDA GPUs reduce the amount of branching (and thus divergence) with predicated execution, i.e., the writeback phase of most instructions can be conditionally disabled by fencing them with a predicate register: if false, the instruction is still executed but does not modify the state of the processor. Conditional execution is automatically introduced by the compiler as a replacement for small, potentially divergent code sequences e.g. in simple if-then-else constructs.

3.2 Memory hierarchy

CUDA devices provide a varied hierarchy of memory areas with different sizes and access times; it is the programmer's responsibility to choose the appropriate usage for each one, also considering access patterns and that no caching is implicitly performed by the hardware.

In addition to a number of 32-bit registers shared by all the threads, each multiprocessor carries an on-chip *shared memory*. Even if slower than registers, shared memory is still significantly fast: its latency can be measured in tens of clock cycles; it is, however, small: our board carries 16 kiB of shared memory per multiprocessor. Shared memory is also banked; multiple accesses to independent banks are carried out simultaneously but conflicts force serialization.

Bulk storage is provided by *global memory*, ranging from hundreds of megabytes up to more than a gigabyte (depending on the card model). The on-board (though not on-chip) global memory is connected to each multiprocessor with a high-bandwidth bus, providing more than 80 Gb/s on our test card. The downside is that every access incurs in a very high latency cost, estimated around 400-600 processor cycles. Latency hiding is one of the reasons for the large number of threads supported by CUDA devices: the hardware scheduler automatically suspends threads that are waiting for the completion of global memory transactions and switches to others that are ready to run.

In order to use efficiently all the available bandwidth it is necessary to perform as few accesses as possible, as wide as the memory bus allows. Since each thread typically accesses small amounts of memory at a time, a hardware controller tries to automatically *coalesce* many smaller memory accesses in fewer, larger transactions at run-time. This is possible only if all the accesses involved respect a well-defined pattern: on newer CUDA devices all the addresses must fall within the same naturally-aligned 256-byte memory area.

CUDA devices provide two further special interfaces to global memory areas under the form of *constant* and *texture memory* that provide the additional benefit of hardware caching. Given their limitations in size (e.g. 64 kiB at most for constant memory) and supported access patterns, they are currently unused by iNFAnt.

3.3 Concurrency and data-sharing model

CUDA devices are intented to be used in scenarios where each thread requires minimal interaction with its siblings: only a subset of the common synchronization and communication primitives is therefore provided.

For any application, the set of active threads is divided into *blocks*: threads from the same block are always scheduled on a specific multiprocessor and communicate through its shared memory; ad-hoc primitives enable atomic read-modify-write cycles. This is the only form of inter-thread communication currently supported by the CUDA model: there are no reliable semantics for concurrent accesses to global memory and threads belonging to different blocks cannot exchange data.

Synchronization works in a similar fashion: CUDA provides primitives for pausing a thread until all the others in the same block have reached the same point in their execution flow. Once again, threads belonging to different blocks cannot interact.

4. INFANT DESIGN

It appears clear from Section 3 that traditional algorithms developed for general-purpose processors are bad matches for the CUDA architecture, often using a small number of threads and paying little attention to memory access patterns. This is even more true for classic automata traversal algorithms: input symbols must be processed sequentially and their randomness can lead to unpredictable branching and irregular access patterns. It appears likely that a good traversal algorithm should be a departure from the traditionally accepted practice.

Given the CUDA architecture and the problem at hand, we have identified the following design guidelines:

1. **Memory bandwidth is abundant**. Reducing the number of per-thread global memory accesses is not a priority if they are fully coalesced and there are enough threads to effectively hide memory latency. Shared memory can be considered fast enough for our purpose without requiring any special considerations.

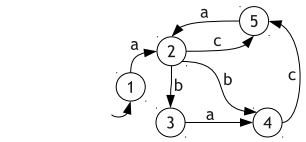
2. **Memory space is scarce**. This is especially true for the shared memory and for registers but global memory should be used carefully as well: although comparatively big, it is common for automata to grow beyond the available amount, even when starting from small regex sets; the ability to store very large automata is also an advantage with multistriding (described in Section 4.3).

3. **Threads are cheap**. In contrast to CPUs, CUDA devices are designed to work best when presented with very large numbers of threads, up to 512 per block, and the maximum number of blocks as supported by the actual GPU considered.

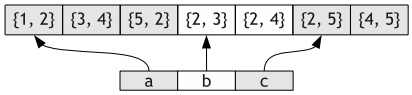
4. **Thread divergence is expensive**. The large number of threads is manageable by the hardware only if all of them execute the same instruction at the same time or if, at worst, the number of possible alternative paths is very small [5]. The program should be structured so that jumps are few and replaceable with predicated execution whenever feasible.

4.1 NFA representation

In order to adhere to our guidelines and in contrast to classic approaches, iNFAnt adopts an internal format for the FSA transition graph that we dubbed *symbol-first* representation: the system keeps a list of $(source, destination)$ tuples, representing transitions, sorted by their triggering symbol. This list can grow very large so it must be stored in a global memory array, together with an ancillary data structure that records the first transition for each symbol, to allow easy random look-ups. As an example, the representation for the automaton in fig. 1(a) is reported in fig. 1(b).



(a) NFA transition graph.



(b) Transition vector.

Figure 1: Symbol-first representation.

The current implementation allocates 16 bits per state label, thus supporting up to 65535 states, which is more than enough for our current workloads that peak at around 6000 - 9000 states. It should be noted that this limitation does not affect the maximum number of transitions, that depends only on global memory availability.

In order to reduce the number of transitions to be stored, and also to speed up execution, iNFAnt adopts a special representation for self-looping states, i.e. those with an outgoing transition to themselves for each symbol of the alphabet. These states are marked as persistent in a dedicated bit-vector and, once reached during a traversal and marked as active, they will never be reset.

Bit-vectors containing current and future active state sets are stored in shared-memory.

4.2 Traversal algorithm

The traversal algorithm follows naturally from the data structure definition. Many packets are batched together and mapped 1:1 to CUDA blocks to be processed in parallel; every thread in each block executes the instructions reported as pseudo-code in fig. 2. State bit-vectors appear with a sv subscript and underlined statements are performed in cooperation by all the threads in a block. More precisely, the copies in lines 1, 5, 13 are performed by assigning each thread a different portion of the bit-vectors involved.

Underlined statements also correspond to synchronization points in the program: after execution, each thread will wait for the others to reach the same point before proceeding.

Parallelism is exploited not only because at any given time multiple blocks are active to process multiple packets but also because for each packet each thread examines a different transition among those pending for the current symbol when running the inner *while* loop (lines 6 – 12). A large number of transitions can be processed in parallel this way and if there are enough threads then the time spent in processing a single symbol will be effectively equal to what would be required for a deterministic automaton.

With regard to access patterns, the traversal algorithm requires global memory for reading the current input symbol and for accessing the transition table: both these ac-

```
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:      end if
12:   end while
13:   current_sv ← future_sv
14: end while
15: return current_sv
```

Figure 2: Traversal algorithm

cesses can be coalesced. All the threads working on the same packet access the same symbol (and offset) at the same time because of synchronization: the card can execute line 3 with a single transaction. Accesses to the transition table are structured so that the N^{th} thread will read the N^{th} offset; if multiple iterations are required, the same pattern is repeated because the stride offset equals the number of threads. These very regular memory patterns are pivotal to exploiting all the available bandwidth and provide big improvements over the almost random patterns that derive from traditional traversal algorithms; even the 'padding' reads that happen on the last iteration of the inner loop if there remain more threads than transitions do not cause a noticeable performance degradation.

The symbol-first representation allows an efficient usage of the available global memory space by storing only useful data, a property that would not be provided by e.g. the classical but potentially very sparse state transition matrix that requires a storage location for each combination of current state and input symbol.

The traversal algorithm, moreover, can be executed with very little divergence among the threads assigned to the same packet: in the current implementation only the conditional choice corresponding to lines 9-10 in fig. 2 can diverge and the compiler is able to handle this case with predicated instructions. Some divergence is possible during initialization or when writing back results, but these phases execute quickly and once per batch of packets, so their overhead is negligible with respect to the total running time of the algorithm.

Finally, it must be noted that the shared memory accesses required in the inner loop for reading the current state vector and setting future states do not follow any predefined pattern. Given the speed of shared memory, its banked structure and the occasional nature of the updates (performed only if a transition actually triggers), this is not expected to be an issue, as confirmed by the results reported in Section 5.1.

4.3 NFA multistriding

An interesting property of the iNFAnt algorithm and data structure is that they easily support multistrided automata. Multistriding is a transformation that repeatedly squares the input alphabet of a state machine and adjusts its transition graph accordingly: intuitively, the alphabet of a 2-strided automaton consists of all the possible pairs of original input symbols and each transition is the composition of 2 adjacent transitions of the original. The transformation required for 2-striding a FSA, documented in [2], can be performed ahead of time and off-line. After multistriding it is possible for the traversal algorithm to consider pairs of symbols at once, thus reducing global execution time.

Squaring has multiple effects, most prominently producing an increase in both transition count and alphabet size. The former is not a major problem if the source automaton is small, but can (and it does, in our tests) quickly lead to memory exhaustion if this is not the case, e.g. when creating DFAs from large rule sets, thus limiting the applicability of the procedure when not working with NFAs. The increase in alphabet size, on the contrary, is particularly troublesome if the procedure is repeated multiple times, as the length of each symbol and the cardinality of the alphabet can quickly approach intractability. In order to avoid this issue iNFAnt performs an alphabet compression step that removes any symbol that does not appear on transition labels and renames all the remaining ones based on equivalence classes. Compression also makes the symbol set dense, allowing simpler data structures to be used for look-ups.

Each multistriding step emits a translation table that (in general) maps pairs of input symbols into a single output symbol: this is possible without an explosion in symbol count because in most cases only a small portion of the possible symbol space is used. In order for a packet to be processed after multistriding, it must first undergo the same translation, a procedure currently performed on the host CPU using a hashed look-up table. The CPU executes this algorithm in pipeline with GPU-based automaton traversal, reducing its run-time impact; the number of symbols to be processed is halved by each rewrite and, by extension, the time required to process each data unit on the GPU is reduced, with no modifications to the traversal algorithm.

4.4 System interface

A major architectural choice is how to interface the iNFAnt engine with other external components, both for the creation of the required data structures and to pass packets to and from the GPU at run-time. The current iNFAnt prototype exposes a simple API that allows loading precomputed NFAs on the graphics card and submitting a batch of packets for processing. Packet copies can be performed through DMA and results are read back in a similar fashion.

While GPU operations such as transfer initiation or kernel launch are not free, they execute quickly and, in most cases, have been found to be of little relevance when compared to the actual time spent in pattern matching.

5. EXPERIMENTAL EVALUATION

iNFAnt has been evaluated by comparing its throughput and memory consumption with those achieved by HFAs, that represent the current state of the art for many purposes by following closely the behavior (and speed) of DFAs on non-troublesome rule sets, while implementing strategies to prevent state space explosions.

The test plan involved 3 regex sets designed to highlight different aspects of the applications under scrutiny. The *http-sig* rule set is composed of 2 regular expressions that recognize specific HTTP headers; the resulting automaton is

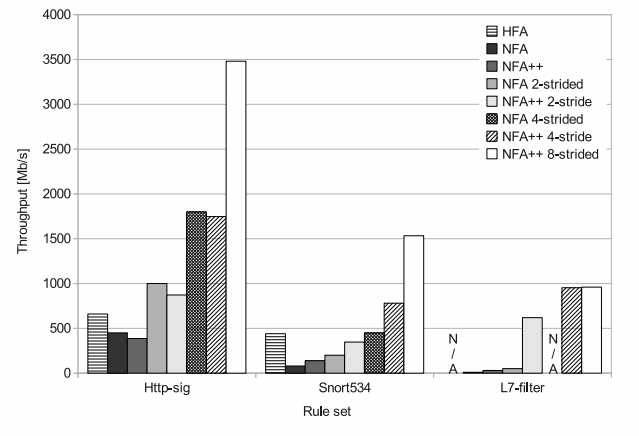


Figure 3: Throughput measurements

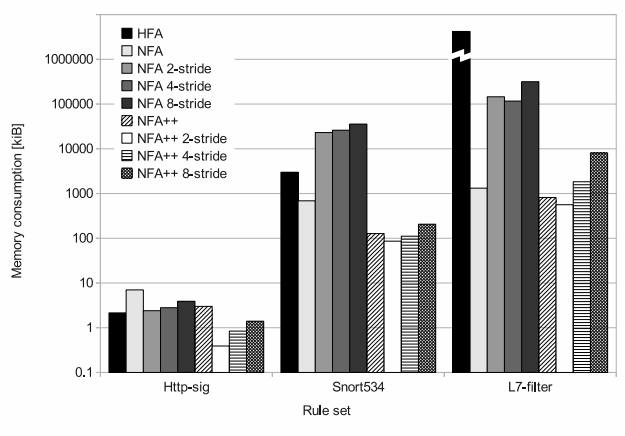


Figure 4: Memory consumption

simple and almost completely linear, posing little challenge both to iNFAnt and HFA and providing baseline results to compare per-byte costs. The *Snort534* set (taken from [3]) consists of 534 regular expressions; it can be divided into subsets that share an initial portion while the tails differ, a structure that makes it a good target for HFAs. Finally, all the protocol signatures from the L7 traffic classifier[4] make up the *L7-filter* set, which is a very complex and irregular test set where no common prefixes or other properties can be exploited. In spite of its limited size (around 120 regexs), the L7-filter is the largest of our test cases in terms of memory occupation, regardless of the form in which it is compiled.

All the tests were performed using a single core of the otherwise-unloaded test machine, a 4-core Xeon machine running at 3 GHz and provided with 4 GiB of RAM; GPU tests were conducted on the same platform equipped with an nVidia GeForce 260 GTX graphics card with 1 GiB of RAM and 27 multiprocessors clocked at 1.24 GHz. All relevant caches (e.g. processor, disk) were warmed by performing unmeasured test runs. As input, a 1 GiB trace of real-world network traffic was used.

The two platforms (GPU card and CPU host system) are significantly different in terms of architecture and specification, thus making their performance not directly comparable. However, they both represent significant examples of commercially available middle-tier hardware. Hence, the throughput measurements reported in the following sections should be regarded as order-of-magnitude estimates of the performance obtainable using commodity hardware devices.

5.1 Pattern-matching throughput

Figure 3 reports the best throughputs obtained for all techniques. In order not to inflate the results, all measurements were performed by taking into account only payload bytes and excluding packet headers (that were not examined). The 'NFA++' data series reports results obtained by enabling the self-looping states optimization described in sec. 4.1. iNFAnt allows the user to set the number of threads per packet and the number of packets submitted to the card in a batch for parallel processing: the best results obtained by exploring the possible configuration space are reported here.

[4]Available at `http://l7-filter.sf.net/`.

As it can be seen, the throughput achieved by non-strided NFAs is comparable to though lower than corresponding HFA results. This can be justified by the higher per-byte traversal cost of NFAs and by the higher instruction execution time of GPUs: even if parallelism reduces the amount of time required to process a single packet, this is not enough to completely compensate for the aforementioned aspects. However, the situation is vastly improved by the introduction of multistriding and the self-loop state optimization, leading to far better throughputs than HFAs.

Given the complexity of the CUDA architecture it is interesting to try to identify the iNFAnt performance bottleneck. Global memory bandwidth, commonly found to be a scalability limitation, is not an issue here: its measured usage is, in most cases, around 20-40 Gb/s, less than the card peak performance (around 80 Gb/s). Shared memory issues can also be ruled out: while it is true that a reduction of its usage would speed execution up (more blocks could be scheduled per multiprocessors) the simulated difference was found to be minimal. Bank conflicts arising from write contention when updating the future state vector are also rare: disabling shared memory updates altogether brings little improvement in run-time performance (1-3% in most cases). Similar considerations also hold for register usage.

Profiling information[5] shows that the vast majority of running time is spent in processing instructions, even if iNFAnt performs very little computation. It is therefore likely that in most cases the bottleneck lies in the relatively large number of instructions to be executed per packet, coupled with the high instruction execution time of GPUs. As with most current traversal algorithms, input symbols must be processed in order, leading to large numbers of iterations in the outer loop of fig. 2. This also explains why multistriding can improve throughput significantly.

5.2 Global memory consumption

Figure 4 shows the amount of global memory required for automaton storage, which is by far the largest data structure used by both the techniques considered; shared memory occupation in iNFAnt is considerably below the maximum amount in all test cases (about 6000 states are required for L7-filter).

[5]Not reported here due to space constraints.

It appears clear that in general the NFAs used by iNFAnt use comparable or less memory than the corresponding HFAs; it is interesting to note that the L7-filter rule set is impossible to compile in HFA form on our test machine, regardless of the provisions built into the HFA model; its column in the chart corresponds to the lower bound of estimated consumption (4 GiB). A direct comparison with DFAs yields even better results for NFAs: besides L7-filter, Snort534 incurs in state space explosion as well. The difference between NFAs and other approaches is exacerbated when considering multistriding: given the increment in size, only the adoption of NFAs makes this technique feasible.

The NFA memory consumption reported must also be considered as a worst-case measurement: the NFAs considered were not in a minimal, canonical form and it might be possible to further reduce their sizes by appropriately modifying the generation process.

5.3 Multistriding and self-loop handling

Both throughput and memory occupation are affected by iNFAnt optimizations. As expected, in most cases multistriding improves run-time performance, mainly because of shortened input packets (in term of symbols), requiring less iterations in the traversal algorithm; the improvements observed are roughly linear with the number of automaton squarings performed, a result consistent with our bottleneck analysis. At the same time, multistriding yields larger automata, mainly because of increased transition counts; this effect is clearly visible in fig. 4. Nevertheless, iNFAnt is effective in dealing with this issue. On one side, as it can be seen from the charts the available amount of global memory is adequate in all cases; on the other side the increase in transition counts is somewhat offset by larger alphabets, making the number of transitions to be examined per symbol grow relatively slowly. As for the rewriting operation itself, in most practical cases it requires less time than automata traversal so its cost can be completely absorbed by pipelining.

Self-looping state optimization, on the contrary, directly reduces transition counts. While obviously not designed to completely counteract the effects of multistriding, the introduction of separate handling for self-looping states proves to be very effective both at reducing the number of transitions stored in global memory (especially with deeper multistriding) and at speeding up execution, once again thanks to lower per-symbol transition counts.

6. CONCLUSIONS AND FUTURE WORKS

This paper presented the design and evaluation of iNFAnt, a novel NFA-based pattern matching engine. iNFAnt is explicitly designed to run on graphical processing units, exploiting the large number of execution cores and the high-bandwidth memory interconnections through its ad-hoc data structure and traversal algorithm; more in detail, the automaton representation and traversal algorithm adopted by iNFAnt match well the CUDA architecture, allowing full coalescing of memory accesses and requiring very little thread divergence.

The adoption of the NFA model allows a significant reduction in memory occupation from the get-go, avoiding state space issues by design and enabling iNFAnt to handle complex rule sets; the optimized handling of self-looping states further reduces memory consumption while at the same time improving run-time performance. Additional free memory, if available, can be traded off for processing speed with the adoption of multistriding, thus effectively counteracting the higher per-byte cost deriving from the nondeterministic model and the high instruction execution time taken by GPUs. Multistriding is especially feasible on the iNFAnt platform because of the lower baseline memory requirements and because the traversal performance depends on the number of transitions per input symbol; other FSA engines, especially if relying on a small alphabet, might be adversely affected by its introduction.

While iNFAnt might not be the first GPU-based pattern matching engine, to the best of our knowledge, it is one of the first to use NFAs to implement a technique specifically designed for graphical processors. In contrast to most approaches ported from general-purpose CPUs, the bottleneck is not memory bandwidth but the execution cores processing speed; higher throughputs could be achieved on the same architecture with more and/or faster execution units.

With regard to future developments, we are planning to perform string rewriting directly on the GPU, thus completely offloading the host CPU: while the task itself is embarrassingly parallel, an efficient implementation of look-up tables on CUDA devices is not. A more thorough evaluation of run-time behavior is also in progress, comparing iNFAnt with more alternative techniques and performing additional scalability tests on more powerful hardware devices.

7. REFERENCES

[1] M. Becchi and P. Crowley. A hybrid finite automaton for practical deep packet inspection. In *proceedings of CoNEXT '07*, pages 1–12, NY, USA, 2007. ACM.

[2] M. Becchi and P. Crowley. Efficient regular expression evaluation: theory to practice. In *proceedings of ANCS '08*, pages 50–59, NY, USA, 2008. ACM.

[3] M. Becchi, C. Wiseman, and P. Crowley. Evaluating regular expression matching engines on network and general purpose processors. In *proceedings of ANCS '09*, NY, USA, 2009. ACM.

[4] F. Kulishov. DFA-based and SIMD NFA-based regular expression matching on Cell BE for fast network traffic filtering. In *proceedings of SIN '09*, pages 123–127, NY, USA, 2009. ACM.

[5] R. Smith, N. Goyal, J. Ormont, K. Sankaralingam, and C. Estan. Evaluating GPUs for network packet signature matching. In *proceedings of ISPASS '09*, pages 175–184, 2009.

[6] G. Szabo, I. Godor, A. Veres, and S. Malomsoky, Sz. and. Molnar. Traffic classification over gbit speed with commodity hardware. In *accepted for publication in IEEE Journal of Communications Software and Systems, 2010, Vol. 5, Num. 3.*, 2010.

[7] G. Vasiliadis, S. Antonatos, M. Polychronakis, E. P. Markatos, and S. Ioannidis. Gnort: High performance network intrusion detection using graphics processors. In *proceedings of RAID '08*, pages 116–134, Berlin, Heidelberg, 2008. Springer-Verlag.

[8] G. Vasiliadis, M. Polychronakis, S. Antonatos, E. P. Markatos, and S. Ioannidis. Regular expression matching on graphics hardware for intrusion detection. In *proceedings of RAID '09*, pages 265–283, Berlin, Heidelberg, 2009. Springer-Verlag.