Performance Evaluation of Regular Expression Matching Engines Across Different Computer Architectures

Vinh Dang*, Jack Wadden*, Mohamed El-Hadedy†, Xiaoping Huang§, Ke Wang*, Mircea Stan*, and Kevin Skadron*

*Center for Automata Processing, University of Virginia, Charlottesville, Virginia 22904, USA
†Coordinated Science Laboratory, University of Illinois, Urbana, Illinois 61801, USA
§Department of Computer Science, Northwestern Polytechnical University, Xi'an, Shaanxi, China

Abstract—Regular expressions are sequences of characters that define search patterns, commonly used in pattern matching with strings. Regular expression matching plays an important role in a variety of applications, such as bioinformatics, network inspection, etc. However, it is a challenging problem because pattern matching is a computationally intensive operation especially when dealing with large data and complex regular expression rulesets. In this work we investigate for a fair comparison of regular expression matching engines across parallel architectures, including multi-core central processing units (CPUs), SIMD-based graphics processing units (GPUs), many-core Intel’s XeonPhi accelerators, field-programmable gate arrays (FPGAs), and Automata Processor (AP). The evaluation involves analyzing their performances with different types of regular expressions and exploring the design spaces of these architectures: “complexity” of regular expressions, the number of regular expressions, and multiple packets processing capability. The work is also extended to other applications that are not natural fits for regular expression.

Keywords—Automata Processor (AP); central processing unit (CPU); graphics processing unit (GPU); field-programmable gate array (FPGA); finite automata; parallel architectures; regular expression matching; Intel’s XeonPhi accelerator

I. INTRODUCTION

Regular expressions are sequences of characters that define search patterns, commonly used in pattern matching with strings. In general, regular expression matching is performed by: (i) converting regular expressions to finite automata (either in deterministic (DFA) on non-deterministic (NFA) form), and (ii) running the automaton on input string(s). While computing a small number of automata can fairly be simple and fast on single central processing units (CPUs), it is a challenging problem when dealing with large data and complex regular expression rulesets. Rulesets of large, complex regular expressions, such as Snort [1] which contains many thousands of rules, can produce large, complex automata, which requires massively high memory bandwidth with low-latency access to compute efficiently. Many researchers have been focusing on accelerating this automata-based computation on a variety of parallel architectures [2]-[5], including multi-core CPUs, SIMD-based graphics processing units (GPUs), many-core Intel’s XeonPhi accelerators, field programmable gate arrays (FPGAs). GPUs and many-core XeonPhi accelerators offer massive data-level parallelism, helping to explore all possible automata transitions simultaneously. Dataflow methods of automata processing on FPGAs can be used to implement automata, and efficiently process a large number of automata transitions in parallel. Recently, Micron has released the Automata Processor (AP) [6], a highly parallel reconfigurable fabric of automata matching elements that can efficiently process complex regular expressions by implementing equivalent NFAs on DRAM.

This paper investigates performance of state-of-the-art regular expression processing engines across five different categories of computer architecture (i.e. CPU, XeonPhi, GPU, FPGA, AP) using identical workloads. The paper aims to evaluate how different types of regular expressions perform on different architectures and identify computational bottlenecks for each of the five named architectures in this application domain. These bottlenecks may help motivate further research into the most efficient regular expression processing methods, especially in heterogeneous environments that involve combinations of these architectures. This work is also extended to a wider range of automata-based applications rather than just limited to only regular-expression-based applications.

II. REGULAR EXPRESSION/AUTOMATA PROCESSING

The regular expression/automata processing engines can be categorized into two main classes: Von Neumann processing and Data-flow processing.

A. Von Neumann Automata Processing

In Von Neumann automata processing, based on active state(s) in the finite automaton, the algorithm looks up appropriate transition rules in memory for each symbol in the input stream and execute those transitions in the automaton. Despite its simplicity, this algorithm relies on high-bandwidth, low-latency memory accesses for good performance. The
automaton can be either NFA or DFA. NFAs allow for an automaton to be in multiple active states and transitions at once. DFAs only allow one active state and one transition per cycle hence they are fast to execute. However, a DFA may require an exponentially larger number of states to represent all possible combinations of NFA states. In this paper, we focus on computation of traditional, theoretical NFA, as a baseline for our comparisons and evaluations.

1) CPU-based Engines: VASim (CPU/XeonPhi)

VASim is a high-performance, open source virtual Automata SIMulator for automata processing research [7]. The core execution architecture of VASim is an optimized version of the classic NFA algorithm described above. VASim only considers automata states that are active, and uses optimized data structures for low-overhead parallel execution of automata. VASim is also parametrically multi-threaded in two dimensions: separate automata can be given to parallel threads, and/or different sections of the input symbol stream can be given to parallel threads. Therefore, many thread contexts can be launched to take advantage of parallel cores in multi-core CPU architectures. While VASim may not be the state-of-the-art, to our knowledge, it is the fastest publicly available NFA simulator, thus we use VASim as the baseline NFA engine on both CPUs and Intel’s XeonPhi many-core co-processor for our evaluations.

2) GPU-based Engine

The most well-known NFA-based regex matching engine on GPUs is the iNFAnt framework, proposed by Cascarano et al [3]. In this framework, the NFA transition tables are organized using a symbol-first format with transitions grouped by their incoming symbols. Since the number of transitions per incoming symbol is not uniform, a supplementary array is used to store locations in the NFA transition table that indicate the first transition of each symbol. Both the NFA transition table and the supplementary array are stored in GPU global memory due to their large size. Bit-vectors are used to represent current and future state vectors and are stored in shared memory of the GPU. The only disadvantage of iNFAnt is that it explores all available transitions on a particular incoming symbol across the entire NFA, while the actual number of active states can be very small.

For our baseline NFA processing engine for the GPU, dubbed iNFAnnt2, we adopt the iNFAnt framework with our own improvements, as well as modifications inspired by Vasiliadis et al. [8], which include: (i) fast accept state recognition by encoding accept states with negative IDs, (ii) multi-byte input symbol fetches, and (iii) storing NFA transition tables in GPU texture memory. We also augmented iNFAnt to report the cycle and rule ID of each matched rule, an important and previously unimplemented feature.

B. Data-flow Automata Processing

1) FPGA-based Engine

FPGAs offer extreme flexibility in the design of regular expression processing engines. We adopt the implementation concept of prior work which exploited the reconfigurable nature of field programmable gate arrays (FPGAs) to lay out automata states and transitions in reconfigurable logic fabrics [5]. More specifically, state machines representing NFA transition tables are implemented in Verilog codes. The codes are then synthesized using Xilinx Vivado Design Suite tool. The FPGA implementation relies on a one-hot encoding of states where each automata state is represented by a flip-flop, and next-state rules are computed using combinational logic. Transitions are also implemented in logic using the FPGA’s programmable routing network. All next-state transitions are computed and propagated in parallel, and the architecture is only limited by state-to-state critical path, and FPGA capacity.

2) AP-based Engine

Micron, leveraging their experience and IP in memory technology, has developed the Automata Processor (AP) [6], a DRAM-based reconfigurable, native-hardware accelerator for non-deterministic finite automata (NFA). The AP implements an NFA using a reconfigurable network of state transition elements (STEs), analogous to NFA states, that all consume a single input stream of 8-bit symbols. If an STE is enabled, and matches the current input symbol, it activates, and propagates enable signals to other STEs via an on-chip routing matrix. All transitions happen in parallel and thus AP performance is always linear in the size of the input symbol stream, and independent of the dynamic activity in the fabric. STEs are capable of single-bit reports, analogous to NFA “accepting states”.

III. PERFORMANCE EVALUATION

We investigate performance of five automata processing engines on five different macro architectures, namely Intel i7-5820K, Intel’s XeonPhi 3120, Maxwell-based NVidia Titan X GPU, Xilinx’s Kintex UltraScale XCKU060-FFVA1156 FPGA (20nm technology node), and Micron’s first-generation Automata Processor (45nm technology node). We select four different sets of automata that represent three major classes of automata applications: regular expression rulesets (regex), string scoring meshes (mesh), programmable widgets (widget). Below we list the four datasets used in our experimental results:

- Brill (regex) [9]: is a set of over 200 Brill tagging rules (26,364 states).
- Protomata (regex) [10]: is a set of 2340 real and randomly generated protein motif signatures (38,251 states).
- Hamming (mesh) [11]: is a set of 93 Hamming distance automata used to calculate the number of mismatches between a randomly generated encoded string and random input sequence (11,254 states).
- Entity Resolution (widget) [12]: is a set of automata designed to identify whether input name sequences match a certain encoded (5,689 states).

Each automaton in the dataset maximizes the resources of a single first-generation Micron AP chip. This allows convenient and fair comparisons between different reconfigurable data-flow architectures and Von Neumann automata processing engines. For example, performance of 1 AP rank (8 parallel AP chips consuming 8 parallel input streams) is trivial to deduce via multiplying the performance of one AP chip by 8.
Similarly, performance of 1 AP board (4 AP ranks consuming 4x8 parallel input streams) is achieved by multiplying the performance of one AP chip by 32.

A. Using the Automata vs Input-level Parallelism Scaling

Automata engines often exploit parallelism in automata and input streams to hide the latency of individual transitions and increase throughput of automata engines. This section explores sensitivity of automata engines and architectures to these two dimensions of parallelism—parallel automata and parallel input streams. Distinct automata can be divided into an equal number of groups (G), and the input stream can be divided into an equal number of sections (S). Thus, we can launch GxS number of CPU threads or GPU thread-blocks to compute an automata-based application in parallel. Due to the page limit, we use the Protomata for this analysis. We pick 4 sets of groups and vary the number of parallel streams on both the VAsim (CPU) and iNFAnt2 (GPU) baseline engines. Results are presented below.

Results from varying automata groups and parallel input streams on VAsim, our baseline CPU automata engine, are shown in Fig. 1a. Protomata is much more responsive to both automata-level and input-level parallelism. This is because Protomata has a small number of automata that have a much greater level of activity than others. The threads that are responsible for these rules run much slower and bottleneck performance. While automata-level parallelism cannot help accelerate these expensive automata, stream-level parallelism can. Thus Protomata performs best with eight parallel automata groups (8), but a larger number of packet streams (12).

Results from varying automata groups and parallel input streams on the iNFAnt2, our GPU automata engine, are shown in Fig. 1b. Unlike the CPU-based engine, the GPU-based engine overwhelmingly favors larger parallel input streams. It performs best when each CUDA thread block operates on all meshes simultaneously and there are more than 560 parallel blocks operating on different sections of the input stream. This highlights the ability of the GPU to hide the latency of any individual memory access by executing an extremely large number of parallel tasks. We observe that 1 group does not universally perform best while 8 groups perform better as the number of parallel streams is increased. This reflects sensitivity to utilization of per GPU stream processor resources like shared-memory and L1 cache. The total performance of the GPU engine relies on a balance of NFA transition table size and stream-level parallelism that is highly application specific.

B. Cross-Architecture Application Evaluation

We evaluate the performance of each NFA automata processing engine over all four datasets. Results are shown in Fig. 2. While this does not represent an absolute ranking of the performance of each architecture, it does represent the current state of the evaluation engines as compared to the AP. For von Neumann engines and architectures (i.e. CPU, XeonPhi, GPU), the “best-effort” performance is recorded. It should be noted that performance tuning of parameters for each von Neumann engine and architecture was considerably time consuming. For each architecture, we explored both dimensions of automata parallelism in attempt find the best performing configuration.
of transitions, active states) while AP’s throughput is immune to these.

Because each automata dataset maximizes the resources of an individual AP chip, we can easily, and fairly, estimate performance of other deployment scenarios of AP chips. Because each AP chip can operate on a separate, parallel portion of the input stream, AP performance is expected to scale perfectly linearly in the real hardware. As an example of this feature, we also include estimated AP Board performance in Fig. 2.

We also estimate the performance of FPGA engine when the full FPGA chip is utilized for processing multiple input streams. The estimation is performed by using the resource utilization reported from the FPGA (single stream, 20nm) implementations. We include these results in the category FPGA (full chip, 20nm) in Fig. 2.

To compare the different architectures in the same technology node, we show the performance of AP Board projected on 20nm node in Fig. 2, assuming linear scaling for clock frequency and square scaling for capacity. The projection is constrained by the bandwidth of PCIe gen3 x8 (7,880 MB/s). The technology-projected performance of AP Board shows better performance than FPGA engines in almost all datasets.

IV. CONCLUSIONS

This paper presented a fair evaluation of automata processing engines on different major computer architectures. Our results indicated that Von Neumann computer architectures for automata processing (multi-core CPU, XeonPhi and GPU) are bottlenecked by memory latency for rule lookup while data-flow processing engines (AP and FPGA) perform best due to its high capacity, its massively parallel execution and its capabilities of processing new input symbol every clock cycle. GPUs and Intel’s XeonPhi can perform well, but exploiting the computational power of SIMD units to compute the irregular parallelism of NFAs is difficult. Thus, these architectures generally benefit from input stream parallelism, rather than automata-level parallelism.

ACKNOWLEDGMENT

This work was supported in part by the Center for Future Architectures Research (C-FAR), one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA.

REFERENCES