RAPID: Accelerating Pattern Search Applications with Reconfigurable Hardware

Kevin Angstadt Jack Wadden Xiaoping Huang[†] Mohamed El-Hadedy[‡]
Westley Weimer Kevin Skadron

University of Virginia, [†]Northwestern Polytechnical University, [‡]University of Illinois at Urbana-Champaign
{angstadt, wadden, weimer, skadron}@virginia.edu, huangxp@nwpu.edu.cn, hadedy@illinois.edu

Abstract

Recent research has demonstrated the efficacy of accelerating textual pattern search tasks using specialized hardware, such as Micron's Automata Processor (AP) and FPGAs, but programming these devices is often challenging for non-hardware-experts. To address this, we present RAPID, a high-level programming language and combined imperative and declarative model for programming pattern-recognition processors. RAPID is clear, maintainable, concise, and efficient both at compile and run time. We discuss the tools and algorithms we have developed and evaluate a suite of RAPID programs against custom, baseline implementations previously demonstrated to be significantly accelerated by specialized hardware. We show that RAPID programs are much shorter in length, are expressible at a higher level of abstraction than their handcrafted counterparts, and yield generated code that is often more compact.

1. Introduction

As analysis of "big-data" sets becomes increasingly more common both in industry [5] and academia (as demonstrated by the emergence of conferences such as IEEE Big Data), there is a growing need for both suitable programming paradigms and high-throughput processing hardware. These analyses can often be re-phrased as pattern-searching problems, in which many searches are conducted against a single stream of data. A pattern defines a sequence of data that should be identified within another collection of data. Hardware technologies, such as FPGAs and Micron's Automata Processor [7], have been demonstrated to accelerate these types of pattern searching tasks in application areas such as natural language processing [22], particle physics [19], machine learning [15, 17, 18], and bioinformatics [4, 11].

While these hardware solutions provide excellent performance for pattern searches, their respective programming interfaces are challenging to use. Current programming models are akin to assembly-level programming on traditional CPU architectures. Consequently, programs written for these accelerators are tedious to develop and challenging to write correctly. Additionally, these low-level representations do not lend themselves well to debugging and maintenance tasks. We argue that *pattern-search problems are better better represented at a higher level of abstraction that is less hardware-dependent.*

In this paper, we present the RAPID programming language, a high-level, C-like language for writing pattern search algorithms. This language maintains the performance benefits of specialized hardware solutions, while providing a clear, concise, and maintainable representation of a search. We attest to RAPID's versatility in representing pattern searches across application domains. We also discuss the tools and techniques we have developed to support execution of RAPID programs on both the Automata Processor and FPGAs. Finally, we present experimental results comparing RAPID programs for real-world applications against state-of-the-art, handcrafted equivalents. Notably, we observe that RAPID programs do not need to be modified to target specific hardware back end; the same program compiles efficiently for both targets.

2. Background and Related Work

Pattern Search Representations. Search patterns are often represented as regular expressions or finite automata, which are equivalent in expressive power (i.e., both can recognize exactly the same class of patterns).

A finite automaton includes of a set of states and a set of transitions defining how the states become active based on symbols observed in an input stream. In a non-deterministic finite automaton (NFA), it is possible to transition to multiple states on the same input symbol.

While capable of specifying search patterns, NFAs are difficult to write and maintain. NFA representations, such as the XML-based Automata Network Markup Language (ANML), are extremely verbose. For example, measuring the pairwise difference of characters between an input string and a five-character string requires 62 lines of ANML to represent [10]. Maintenance tasks on this code are also cumbersome: modifying the automaton to compare against a string of length 12 requires modification of 65% of the code. NFAs

```
macro frequent (String set, Counter cnt) {
  foreach(char c : set) {
    while(input() != c);
  }
  cnt.count();
}

network (String[] set) {
  some(String s : set) {
    Counter cnt;
    whenever(START_OF_INPUT == input())
      frequent(s,cnt);
    if (cnt > 128)
      report;
  }
}
```

Figure 1. Example RAPID program implementing association rule mining.

can be challenging and tedious to write correctly, especially for developers lacking familiarity with automata theory. In research areas such as program verification, the task of specifying automata is automated [1].

Regular expressions are another common option for representing a search pattern; however, these also suffer from similar maintainability challenges. For many of our target applications, such as motif searches, particle tracking, and rule mining, the regular expression representing the search is non-intuitive and may simply be an exhaustive enumeration of all possible strings that should be matched. Additionally, programming of regular expressions can be extremely error-prone due to variations in regular expression syntax, which leads to high rates of runtime exceptions [13].

Hardware Support for Pattern Searches. The problem of accelerating pattern searches has a long history [6, 8, 21]. Recently, there has been been a renewed focus on accelerating pattern-recognition problems using specialized hardware. Some examples are IBM's PowerEN Processor [9], the Titan IC RXP Regular eXpression Processor [14], and the Automata Processor (AP) [7]. The PowerEN processor contains special accelerators alongside traditional cores to allow for improved regular expression performance. The Titan IC processor employs a large lookup table to accelerate regular expression matching. Rather than directly processing a regular expression, the AP executes NFAs using a combination of a modified SDRAM memory array and a reconfigurable routing matrix. Researchers have also developed FPGA-based regular expression and automata engines, which have accelerated these tasks [12, 20]. Our work targets both the AP and FPGAs. In particular, we argue that a pattern-search language should be *hardware agnostic*, with efficient compilation to a variety of hardware targets.

3. RAPID Language Overview

In this section, we present an overview of the RAPID programming language. For a more detailed description, see Angstadt et al. [2]. RAPID is a concise and maintainable hardware-agnostic, high-level programming language for defining highly-parallel pattern searches against a stream of data. Programs consist of one or more *macros* and one *network*, written in a combination of imperative and declarative styles. A *macro* uses sequential control flow to define an algorithm for matching patterns in an input data stream. The *network* contains a list of macros that are instantiated in parallel, allowing for simultaneous recognition of many patterns in data streams. Macros and networks provide a programming paradigm that supports a high-level notion of pattern searches and also maps naturally to the underlying computational models of the AP and FPGAs.

Programs passively observe an input symbol stream, abstractly represented as a stream of 8-bit characters and accessed through calls to the `input()` function. A call to this function returns the head of the data stream and acts as a synchronization point in the program. All active computation executes up to the next call to `input()` and then receives the same symbol from the input stream. When a pattern is identified in the input stream, RAPID programs generates a *report* event, which captures the offset in the input stream as well as the macro triggering the report event.

While the network of a RAPID program provides coarse-grained parallelism, finer-grained control of parallelism is achieved through use of three parallel control structures: `either/orelse`, `some`, and `whenever`. These statements allow for both static and dynamic spawning of computation to compare against multiple character sequences in parallel. Additionally, these statements provide support for *sliding window* searches, in which the program is searching for a pattern that could begin on any character of the input stream or following a set sequence of characters.

An example RAPID program is presented in Figure 1. The example program implements association rule mining (also referred to as frequent itemset mining). Candidate association rules are passed in to the *network* as elements of the `set` array. All candidates are checked in parallel against the input data stream after being triggered by a sliding window search for a reserved `START_OF_INPUT` symbol. Candidate rules that occur frequently enough within the input stream trigger reports, which can be used during post processing for filtering.

4. Compilation Workflow

We employ a workflow of several tools to target FPGAs and the Automata Processor as depicted in Figure 2. Each stage in this pipeline represents an explicit *lowering* of the programmatic representation beginning with a high-level RAPID program and producing a binary file. We describe each stage of this process below.

RAPID Compiler. The compilation process begins with an input RAPID program. We have developed techniques for converting such programs into functionally equivalent NFAs

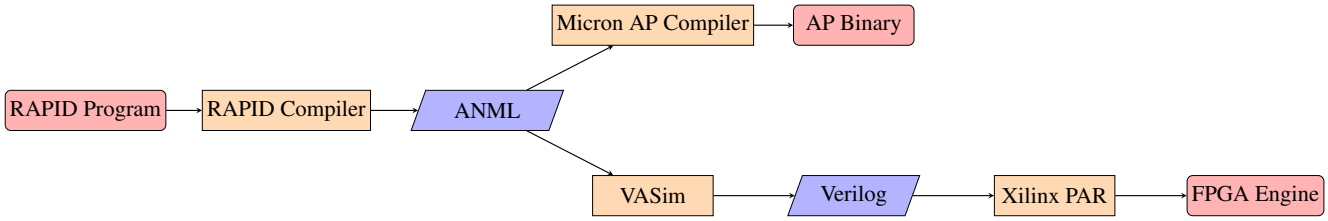

Figure 2. Compilation pipeline for RAPID programs.

(i.e. automata that are able to recognize the same symbol patterns) [2]. We perform this transformation recursively much in the same that a regular expression may be converted into a non-deterministic finite automaton. Comparisons against the input stream are encoded into the states of the design, and higher-level constructs such as loops, parallel control structures, and conditionals are transformed into the connections between these states. This produces a collection of NFAs encoded in an ANML specification that can then be further processed to execute on an Automata Processor or FPGA.

Targeting the Automata Processor. Micron provides a proprietary tool for converting ANML specifications into a binary image that is loadable onto an Automata Processor. This tool places and routes the NFAs onto the hardware states and reconfigurable routing mesh of the processor. We use this tool directly to synthesize ANML for the AP.

Targeting FPGAs. We have developed and collected a set of algorithms for optimizing and transforming automata designs. These algorithms are implemented in VASim, a tool we have created to facilitate automata research and experimentation [16]. VASim additionally supports multi-threaded execution of automata on CPUs. We use this tool to first optimize the input automaton from the RAPID compiler using what is known as *common prefix collapsing* [3]. This process merges states that match the same input symbols, beginning with the starting states, producing a functionally-equivalent NFA with fewer states. In our Brill tagging benchmark, for example, left minimization results in a 57% reduction in the number of states.

Using this optimized NFA, VASim then transforms the design into a Verilog hardware description. Our tool generates a module with inputs for clock, reset, and an 8-bit input symbol and outputs for report events. Within the module, activations of states in the automaton are stored in registers, which are updated on every clock cycle. A state becomes *active* if it is enabled (a state with an incident edge to the current state is active) and the current input symbol matches. Using this update rule, it is possible to execute the NFA directly in hardware. Collection and exporting of report events is left as future work. Finally, we target Xilinx FPGAs by synthesizing the hardware description produced by VASim.

Table 1. Comparison between RAPID and hand-crafted NFAs (ANML) with respect to lines of code (LOC).

Benchmark	RAPID LOC	ANML LOC	Percent Reduction
ARM	18	301	94.02%
Brill	688	9698	92.91%
Exact	14	193	92.75%
Gappy	30	2155	98.61%
MOTOMATA	34	587	94.21%

Table 2. Space utilization on AP and FPGA targets. Lower values for AP States, FPGA LUTs and FPGA Registers indicate a smaller footprint; lower values for AP Mean BR Allocation indicate less stress on the routing network.

Benchmark		AP States	AP Mean BR Alloc.	FPGA LUTs	FPGA Registers
ARM	H	58	20.8%	73	76
	R	56	20.8%	83	65
Brill	H	1,514	65.4%	201	1483
	R	1,429	60.6%	358	1360
Exact	H	27	4.2%	6	25
	R	27	4.2%	28	27
Gappy	H	123	77.1%	73	123
	R	399	70.8%	52	399
MOTOMATA	H	149	75.0%	114	148
	R	72	75.0%	85	60

H – Handcrafted R – RAPID

5. Experimental Results

We evaluate RAPID against hand-crafted automata designs for five real-world applications, which were selected based upon previous research demonstrating significant hardware acceleration with the automata paradigm. The benchmarks are: association rule mining (*ARM*) [18], Brill part of speech tagging (*Brill*) [22], exact- and gappy-match DNA alignment (*Exact* and *Gappy*) [4] and planted motif search (*MOTOMATA*) [11]. For each benchmark, we chose an instance size representative of a real-world problem as specified by authors of the previous research.

Table 1 presents an evaluation of the conciseness of the RAPID language. We compare the lines of code needed to represent the given application to a hand-crafted NFA written in ANML. We choose ANML for representing NFAs because of its use in the AP toolchain and because each

line roughly corresponds to a single state or transition. For the five benchmarks evaluated, pattern searches written in RAPID reduces the program text by 92%–98%.

The RAPID compiler also produces automata that are as space efficient as their hand-crafted counterparts. Because automata runtimes are linearly dependent on the length of the input, performance depends on the number of automata that can be executed in parallel (a smaller footprint allows for more automata to process in parallel). Table 2 compares the physical space (AP states) needed to execute a single widget of each benchmark on the AP. The current generation AP has approximately 1.5 million states. A lower number of states means that more widgets can be loaded onto the AP and run in parallel, thereby increasing device throughput. AP Mean BR allocation is a metric that approximates of the routing complexity of the design. Lower numbers indicate less congestion. These results demonstrate that describing pattern searches at a higher level of abstraction (i.e., in RAPID) does not result in significant space overheads on the AP. In fact, for four of the five benchmarks, RAPID version is at least as compact as its hand-crafted counterpart.

We also evaluate the space efficiency of the FPGA engines our tools produce. We synthesize our designs for a Xilinx Kintex UltraScale XCKU060. Table 2 also lists the number of LUTs and registers needed to implement the hardware description of the pattern search. Lower numbers indicate smaller footprints for the circuits, which allows for more widgets to be run in parallel on the FPGA. We note that, as with the AP results, RAPID programs do not incur significant space overheads on the FPGA. This initial result indicates that RAPID programs are hardware agnostic (i.e., the same RAPID program can be efficiently executed on both the AP and FPGAs). We leave a complete timing analysis and comparison with other FPGA engines to empirically validate this claim for future work.

6. Conclusions

In this short paper we discuss RAPID, a high-level programming language for specifying pattern searches. Our experimental evaluation of RAPID using five real-world applications demonstrate that RAPID programs are concise, maintainable, and efficient on both the AP and FPGAs. We are continuing to develop our toolchain and explore additional optimizations to support more efficient execution on FPGAs as well as other architectures.

Acknowledgments

We acknowledge the partial support of the NSF (CCF-0954024, CCF-1116289, CDI-1124931); Air Force (FA8750-15-2-0075); Virginia Commonwealth Fellowship; Jefferson Scholars Foundation; the ARCS Foundation; and C-FAR, one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA. We would like to thank Micron Technology for their support with programming the AP.

References

[1] R. Alur, P. Černý, P. Madhusudan, and W. Nam. Synthesis of interface specifications for Java classes. In *POPL '05*, pages 98–109, 2005.

[2] K. Angstadt, W. Weimer, and K. Skadron. Rapid programming of pattern-recognition processors. In *ASPLOS '16*, pages 593–605, 2016.

[3] M. Becchi and P. Crowley. Efficient regular expression evaluation: Theory to practice. In *Proceedings of Architectures for Networking and Communications Systems*, pages 50–59, 2008.

[4] C. Bo, K. Wang, Y. Qi, and K. Skadron. String kernel testing acceleration using the Micron Automata Processor. In *Workshop on Computer Architecture for Machine Learning*, 2015.

[5] Capgemini. Big & fast data : The rise of insight-driven business. `http://www.capgemini.com/resource-file-access/resource/pdf/big_fast_data_the_rise_of_insight-driven_business-report.pdf`, 2015.

[6] H. D. Cheng and K. S. Fu. VLSI architectures for string matching and pattern matching. *Pattern Recognition*, 20(1):125–144, 1987.

[7] P. Dlugosch, D. Brown, P. Glendenning, M. Leventhal, and H. Noyes. An efficient and scalable semiconductor architecture for parallel automata processing. *IEEE TPDP*, 25(12):3088–3098, 2014.

[8] A. Halaas, B. Svingen, M. Nedland, P. Sætrom, O. Snøve, Jr., and O. R. Birkeland. A recursive MISD architecture for pattern matching. *IEEE TVLSI*, 12(7):727–734, 2004.

[9] A. Krishna, T. Heil, N. Lindberg, F. Toussi, and S. VanderWiel. Hardware acceleration in the IBM PowerEN processor: Architecture and performance. In *PACT '12*, pages 389–400, 2012.

[10] Micron Technoloy. Calculating Hamming distance. `http://www.micronautomata.com/documentation/cookbook/c_hamming_distance.html`.

[11] I. Roy and S. Aluru. Finding motifs in biological sequences using the Micron Automata Processor. In *IPDPS '14*, pages 415–424, 2014.

[12] I. Sourdis, J. Bispo, J. M. P. Cardoso, and S. Vassiliadis. Regular expression matching in reconfigurable hardware. *Journal of Signal Processing Systems*, 51(1):99–121, 2008.

[13] E. Spishak, W. Dietl, and M. D. Ernst. A type system for regular expressions. In *Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs*, pages 20–26, 2012.

[14] Titan IC Systems. RXP regular eXpression processor soft IP. `http://titanicsystems.com/Products/Regular-eXpression-Processor-(RXP)`.

[15] T. Tracy, Y. Fu, I. Roy, E. Jonas, and P. Glendenning. Towards machine learning on the Automata Processor. In *Proceedings of ISC High Performance Computing*, pages 200–218, 2016.

[16] J. Wadden and K. Skadron. VASim: An open virtual automata simulator for automata processing application and architecture research. Technical Report CS2016-03, University of Virginia, 2016.

[17] K. Wang, E. Sadredini, and K. Skadron. Sequential pattern mining with the Micron Automata Processor. In *Proceedings of the ACM Intl. Conf. on Computing Frontiers*, pages 135–144, New York, NY, USA, 2016. ACM.

[18] K. Wang, M. Stan, and K. Skadron. Association rule mining with the Micron Automata Processor. In *IPDPS '15*, 2015.

[19] M. H. Wang, G. Cancelo, C. Green, D. Guo, K. Wang, and T. Zmuda. Using the Automata Processor for fast pattern recognition in high energy physics experiments - a proof of concept. *Nuclear Instruments and Methods in Physics Research*, 2016, to appear.

[20] X. Wang. Techniques for efficient regular expression matching across hardware architectures. Master's thesis, University of Missouri-Columbia, 2014.

[21] H. Yamada, M. Hirata, H. Nagai, and K. Takahashi. A high-speed string-search engine. *IEEE JSSC*, 22(5):829–834, 1987.

[22] K. Zhou, J. J. Fox, K. Wang, D. E. Brown, and K. Skadron. Brill tagging on the Micron Automata Processor. In *Proceedings of the 9th IEEE Intl. Conf. on Semantic Computing*, pages 236–239, 2015.