RAPID Programming of Pattern-Recognition Processors

Kevin Angstadt   Westley Weimer   Kevin Skadron

Department of Computer Science
University of Virginia
Charlottesville, VA 22904-4740
{angstadt,weimer,skadron}@cs.virginia.edu

Abstract

We present RAPID, a high-level programming language and combined imperative and declarative model for programming pattern-recognition processors, such as Micron's Automata Processor (AP). The AP is a novel, non-Von Neumann architecture for direct execution of non-deterministic finite automata (NFAs), and has been demonstrated to provide substantial speedup for a variety of data-processing applications. RAPID is clear, maintainable, concise, and efficient both at compile and run time. Language features, such as code abstraction and parallel control structures, map well to pattern-matching problems, providing clarity and maintainability. For generation of efficient runtime code, we present algorithms to convert RAPID programs into finite automata. Further, we introduce a tessellation technique for configuring the AP, which significantly reduces compile time, increases programmer productivity, and improves maintainability. We evaluate five RAPID programs against custom, baseline implementations previously demonstrated to be significantly accelerated by the AP. We find 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. In addition, our tessellation technique for configuring the AP has comparable device utilization to, and results in compilation that is up to four orders of magnitude faster than, current solutions.

1.   Introduction

Big-data research is becoming increasingly common in both industry and also academia. In a recent survey of around 1,000 senior decision-makers from nine industries and ten countries, 70% of respondents consider their business's ability to exploit big data critical to their future success [5]. Additionally, the Computer Sciences Corporation reports that the amount of data being generated by individuals and companies will be 44 times greater in 2020 than it was in 2009 [8].

Collected data is often analyzed in a multitude of different ways, and many algorithms in areas such as data-mining, bioinformatics, deep packet analysis, and spam filtering require identification of exact or near-match character patterns. A pattern defines a sequence of data that should be found within another collection of data. For example, a pattern could be all DNA sequences within a Hamming distance of four from "ATCGAC" or all six-letter strings beginning with "se". Conducting such searches through large datasets demands good support from both hardware and software. Current processor technologies are not well-suited for these tasks: single-threaded processing most naturally identifies a single pattern at a time, and therefore requires multiple passes through the data; vector or SIMD processors suffer from inherent branch divergence when analyzing data for multiple patterns; and multi-core processors and clusters often execute far fewer threads than there are patterns to be identified.

Hardware designers are exploring new processing technologies to accelerate pattern identification. One proposed solution is the Automata Processor (AP) [10] by Micron, a non-Von Neumann architecture for direct hardware simulation of non-deterministic finite automata (NFAs). Patterns are encoded as NFAs, which are loaded into a reconfigurable lattice on the processor. Data are then streamed to the processor and the NFAs are executed in parallel. The AP is well-suited for problems that are highly parallel with disjoint or inexact comparisons, such as analyses requiring the identification of multiple patterns against a single data stream. Previous work has demonstrated significant speedups on the AP for several data analysis and pattern-recognition problems, such as biological motif search [16], association rule mining [19], and Brill tagging in natural language processing [21].

The AP, however, is currently challenging to program. Programming consists of specifying NFAs using an XML-based language, which requires both knowledge of automata theory and also the non-standard architecture. This is akin

to assembly-level programming on a traditional, Von Neumann architecture. High-level language bindings exist for this XML-based language, but are similarly challenging to use. Regular expressions are a common means for defining NFAs, but their use with the AP also has drawbacks. For problems accelerated by the AP, the regular expression representation is often an exhaustive enumeration of all accepting data sequences. This is difficult to maintain and places an unnecessary burden on the developer, as described further in Section 2.

This paper presents RAPID, a high-level language that maintains the performance benefits of pattern-recognition processors while also providing concise, clear, maintainable, and efficient representations of pattern-identification algorithms. Programs consist of one or more *macros* and a *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. Macros in RAPID are similar to C-style macros and macros in low-level Automata Processor programs. The *network* contains a list of macros that are instantiated in parallel, allowing for simultaneous recognition of many patters in data streams. Macros and networks provide a programming abstraction that maps naturally to both pattern-matching problems and also the computational model of the Automata Processor while providing a familiar structure akin to functions or procedures.

We introduce three parallel control structures to facilitate common pattern-matching tasks. These allow the concise specification of multiple, simultaneous comparisons against a single data stream and provide high-level support for variable-offset sliding window comparisons that are integral to many pattern-recognition problems.

We also present algorithms for converting RAPID programs into NFAs for execution with the Automata Processor. While code generation from RAPID for other pattern-recognition processors and CPUs is possible, we choose to focus on the AP because of recent promising results [4, 16, 19, 21] and the overall flexibility of the architecture. These studies have reported speedups ranging from 8x–4000x over single-threaded CPU applications. Acceleration of RAPID programs is achieved via *staged computation*. Imperative statements in the code are executed at compile time to aid in generating finite automata, while declarative statements related to the input data stream are then executed on the Automata Processor.

The AP hardware has a finite number of resources, divided into configurable regions. By increasing the utilization of these resources, more NFAs can be loaded, thereby allowing for increased parallel exploration of the data stream. We propose an *auto-tuning tessellation* optimization to increase the density of NFAs loaded into the AP while reducing compile time. Often, the generated finite automata of a RAPID program consist of a repeated NFA design for exploring vari-

ations of a pattern. Our optimization determines the maximum quantity of this design that can occupy the smallest programmable portion of the AP and generates a final automaton accordingly. At runtime, this structure can be *tiled* (loaded into multiple, programmable regions) onto the AP by making use of block-level configuration before execution of the search. Automation of this process saves the developer time and reduces the number of possible programming mistakes, while increasing the overall resource utilization and throughput of the AP.

We evaluate the efficiency of compiled RAPID programs against handcrafted equivalents, measuring program size and resource utilization. These programs are based on real-world applications that have previously been show to have significant speedups when executed on the Automata Processor. This paper makes the following contributions:

- the RAPID programming language, a high-level language for programming pattern identification processors

- a set of algorithms for converting RAPID programs into non-deterministic finite automata

- an auto-tuning tessellation optimization for block-level configuration of the AP, which maintains runtime efficiency and significantly reduces compilation times

- experimental evaluation of the RAPID language against hand-crafted applications demonstrating improved density of loaded NFAs and up to five orders of magnitude faster compilation times when using our tessellation optimization

2. Background and Related Work

Pattern-Recognition Processors.   There have been many proposed hardware approaches for accelerating pattern-matching problems [7, 11, 12, 14, 20]. Recently, there has been been a renewed focus on accelerating pattern-recognition problems using devices such as IBM's PowerEN Processor [13], the Titan IC RXP Regular eXpression Processor [18], and the Automata Processor (AP) [10]. 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. The device is a MISD processor according to Flynn's Taxonomy. In this paper we choose to target the AP and leave research into alternate hardware architectures for future work.

Low-Level Automata Programming.   The primary programming model for the AP requires developers to design NFAs using an XML-based language called the Automata Network Markup Language (ANML). The ANML definition is passed to placement and routing tools, which generate a

binary image of the automaton. This binary image is used to configure AP components to directly execute the NFAs. The language allows for very fine-grained control, but is verbose and not conducive to maintenance. For example, measuring Hamming distance, the number of differences in corresponding characters between two strings, between a five character string and the input data stream requires 62 lines of code [15]. To modify this code for Hamming distance comparisons with a string of length 12 requires 40 lines to be modified or inserted (i.e., 65% of the code in this simple example must be modified to make this change). Modifying the code to support even larger strings requires the modification of an increasing percentage of the code.

Although GUI-based design tools for developing automata exist, there is still an additional level of abstraction the developer must provide. Instead of directly representing the pattern to be matched, the developer must instead provide a finite automaton. The conversion from pattern to automaton is a conceptual burden placed on the developer. Finite automata can be challenging and tedious to design and can feel non-intuitive to developers lacking familiarity with automata theory. In program verification research, for example, generation of finite automata that represent specifications is automated because generating correct designs is difficult [1], and debugging of these automata is also challenging [2].

Additionally, ANML requires developers to have intimate knowledge of the processor's architecture to design automata that meet hardware constraints. Consequently, changes in newer generations of the hardware could hinder productivity and portability.

Alternatively, developers could program the processor using regular expressions, which abstractly represent their equivalent automata; however, this also has its drawbacks. For many problems, such as motif search or association rule mining, the regular expression representing the search is non-intuitive or may enumerate all possible variations of a given pattern that should be matched. An additional limitation with these programming models is that the patterns to be matched must be known at compile time; i.e., a *specific instance* of a problem must be defined and then compiled.

Languages for Streaming Applications. Streaming applications process a sequence of data that is received in real time. Common examples of these applications include radio receivers and software routers. The AP can be classified as a streaming processor because the hardware detects patterns in real time as data is input.

Languages for streaming applications, such as StreamIt [17], have been studied in great detail. StreamIt provides structures for stream pipelining, splitting and joining, and feedback loops. StreamIt objects may peek and pop from the input stream, store input, and perform computations before outputting a result. The AP architecture, however, does not readily admit to this computational model. Finite automata have no inherent memory, and cannot generally peek at the input stream. Many of the operations allowed by StreamIt are thus not applicable in this domain, and it is not evident how to extend the StreamIt model to describe complex automata nor non-deterministic execution. Additionally, data and control are treated differently in StreamIt and RAPID. A StreamIt program specifies a stream graph: data always enters the program and is transformed and passed downstream until reaching an output. In RAPID, each instruction describes the next step to be taken to identify a pattern: stream data enters the program in the location(s) where the program is currently active, causing control to shift to another statement in the program. Ultimately, StreamIt and RAPID target different architectures and abstractions, and are not directly comparable (e.g., it is not clear how to compile StreamIt programs for the Automata Processor).

Non-Deterministic Languages. Non-determinism is a useful formalism for identifying patterns in parallel within a data stream. In a state machine, non-determinism arises when multiple states are active simultaneously, effectively allowing for parallel exploration of the data stream. Several existing languages contain non-deterministic control structures to facilitate these types of operations.

Dijkstra's Guarded Command Language [9] introduces non-deterministic alternative and repetitive constructs. Each construct is predicated with a boolean "guard" that must be true for the encapsulated statements to execute. The alternative construct chooses one command for which the guard is satisfies and executes it. In the repetitive construct, the program loops, choosing one command with a satisfied guard to execute, until no guards are satisfied. Rather than proposing a concrete language, the Guarded Command Language presents guiding formalisms for supporting non-determinism. RAPID provides similar constructs, but a notion of parallel exploration is incorporated directly into the semantics, allowing RAPID to be more concise than the Guarded Command Language when processing stream or pattern data.

An additional non-deterministic programming language is Alma-0 [3], a declarative extension of Modula-2. Alma-0 supports the use of boolean expressions as statements, an ORELSE statement allowing for execution of multiple paths through the program, and a SOME statement that is the non-deterministic dual of the FOR statement. While RAPID also treats boolean expressions as statements (see Section 3.1), it differs from Alma-0 in the computational model supported by the language's semantics. In Alma-0, the ORELSE and SOME are defined via backtracking. Execution is single threaded: when an ORELSE statement or a SOME statement is encountered, the program will choose a single option to execute. If an exploration fails, the program backtracks to the last choice point, restoring all program state, and attempts a different option. Rather than choosing a single option to

```
1  macro hamming_distance (String s, int d) {
2      Counter cnt;
3      foreach (char c : s)
4          if(c != input()) cnt.count();
5      cnt <= d;
6      report;
7  }
8  network (String[] comparisons) {
9      some(String s : comparisons)
10         hamming_distance(s,5);
11 }
```

Figure 1: A RAPID program for computing Hamming distances

explore and backtracking if computation fails, RAPID programs explore all paths in parallel.

3. The RAPID Language

RAPID allows developers to write concise, clear, and maintainable algorithms for use on pattern-recognition processors, such as the Automata Processor. In particular, RAPID supports searching a stream of data for many patterns in parallel. Programs are written in a combined imperative and declarative style using a C-like syntax. In this section, we present a high-level overview of the control structures and data representations in the RAPID programming language.

3.1 Program Structure

Macros and Networks. Rapid programs consist of one or more *macros* and a *network*. The basic unit of computation in a RAPID program is a *macro*, which define reusable pattern-matching algorithms. Macros in RAPID share similarities with both C-style macros and ANML macros, allowing code to be written once and then used as a "rubber stamp". RAPID macros admit more customized usage than their namesakes in C and ANML; the same macro can generate all designs for a particular problem.

Statements within a macro are executed sequentially and define actions that should be taken to identify a pattern. To facilitate the description of these actions, RAPID provides several control structures, including `if` statements, `while` loops, and `foreach` loops. Unlike some languages, we guarantee in-order traversal when iterating with a `foreach` loop. The language also provides parallel control structures useful for pattern-matching, which we describe later in this section.

Additionally, macros can instantiate other macros. When a macro is called, control shifts to the called macro; all of its statements are executed, and then control returns to the calling macro. While the macro code defines how to identify a pattern in the input stream, the macro parameters can specify the particular characters to match, allowing for comparisons of varying lengths. Consider the macro in Figure 1, which performs a Hamming distance computation between a string parameter, `s`, and the input stream. Changing from comparison against a string of length five to a string of length twelve only requires passing a different string argument to the macro. As noted previously in Section 2, more than half of the code in the ANML model must be modified to make an identical change.

The *network* represents the highest level of pattern-matching within a RAPID program. Statements within a network definition are executed in parallel. The most common use of the network is to define a collection of macros for instantiation, which are executed in parallel at runtime to identify patterns in the input data stream. The network may also have parameters which specify certain values at runtime. For example, Figure 1 contains a RAPID program that computes the Hamming distance for a number of given strings and reports on input within a distance of five. In this example, the network is parameterized on an array of strings, which is used at runtime to specify the comparisons being made.

Reporting. RAPID programs passively observe the input data stream; they cannot modify the stream in any manner. Programs can indicate interesting regions within the stream by using the `report` statement, which generates a *report event*. These events provide the offset in the input data stream at which the report occurred and additional identifying meta data, such as the reporting macro. For the program in Figure 1, reports indicate offsets where the input stream is within a Hamming distance of five from the strings in `comparisons`.

Boolean Expressions as Statements. Inspection of the input data stream is central to the RAPID programming model. Often, pattern identification algorithms only continue if a certain sequence of characters is detected. RAPID provides concise support for this common domain idiom by allowing boolean expressions whenever full statements are allowed. These declarative assertions terminate the thread of computation if the expression returns `false`. Line 5 in Figure 1 illustrates this usage.

3.2 Types and Data in RAPID

There are five primary data types in RAPID: `char`, `int`, `bool`, `String`, and `Counter`. Both `String` and `Counter` are lightweight objects, while the remaining three are primitive types. Additionally, there is support for nested arrays of these types.

In RAPID, pattern-matching occurs in a stream of a characters. Therefore, the language provides the `char` primitive type for interacting with input data. The input data stream, however, is a stream of bits and does not need to be interpreted as characters. To support this, a `char` may also store escaped hexadecimal values. RAPID also defines two character constants, which represent special symbols in the input stream: `ALL_INPUT` and `START_OF_INPUT`. The former represents any symbol within the input and the latter is a reserved symbol for indicating the start of data. For example,

```
1  Counter cnt;
2  foreach(char c : "rapid") {
3      if( c == input() ) cnt.count();
4  }
5  if( cnt >= 3 ) report;
```

Figure 2: The above code counts the number of characters matched in "`rapid`" and reports if the count is at least three

if the input data stream consists of the flattening of an array, the entries would be concatenated into a stream, separated by the `START_OF_INPUT` symbol.

A `Counter` is an abstract representation of a saturating up-counter. Upon instantiation, a counter's value is initialized to zero. Counters provide two functions: `reset()` and `count()`, which set the value to zero and increment by one respectively. Although a program does not have access to the internal value of the counter, it is possible to check its value against integers.

Figure 2 demonstrates the usage of counters and interacting with the input stream. The `foreach` loop iterates over each character in the string "`rapid`" sequentially. If that character matches the next character from the input stream, the counter is incremented. After iterating over the entire string, the program checks if the counter is at least three and reports if so. For example, if the stream contained "`tepid`", the count would be three, and there would be a report, but "`party`" results in a count of one and no report.

The input data stream in RAPID is privileged and is accessed via the `input()` function. A call to this function returns a single character from the head of the data stream. Access to the input data is destructive—no peeking or insertion is allowed. All portions of a RAPID program executing in parallel receive the same character from the input stream. For example, if the stream contains "`abcd...`", `input()` would return 'a' to all active threads of computation, and the stream would now contain "`bcd...`".

RAPID's design represents the input stream as a special function rather than as a special indexed array. This is for conceptual clarity: arrays afford a notion of random access into the stored data, while pattern-recognition processors support sequential access to an ordered sequential data stream. Global input access is intentionally similar to C's "fgetc" rather than "fread/fseek" or "mmap".

3.3 Parallel Control Structures

In pattern-matching problems, it is often useful to explore multiple possibilities in parallel. For example, a spam filter may wish to check for many black-listed subject lines simultaneously, or a gene aligner may begin matching a sequence at any point in the input stream. To facilitate such operations, RAPID provides both the network environment and also parallel control structures. Networks, as described previously, allow for parallelism at the macro level, which is

```
1  either {
2      hamming_distance(s,d); //hamming distance
3      'y' == input();        //next input is 'y'
4      report;                //report candidate
5  } orelse {
6      while('y' != input()); //consume until 'y'
7  }
```

Figure 3: An example usage of an either/orelse statement

useful for checking several patterns in tandem. The parallel control structures (`either/orelse`, `some`, and `whenever`) provide finer-grain control over parallel operations.

Either/Orelse Statements. This structure provides basic support for parallel exploration in a pattern-matching algorithm. An `either/orelse` statement consists of two or more blocks, which allows for an arbitrary, static number of parallel computations. Computation splits when an `either/orelse` statement is encountered during execution, and each of the blocks is executed in parallel. When the end of a block is reached, computation continues with the next statement in the program. No blocking or joining occurs, meaning that different paths in the `either/orelse` statement may begin executing the following statement at different times. This behavior is desirable because it allows for the matching of different length patterns containing the same suffix.

As an example usage of the `either/orelse` statement, consider the code fragment in Figure 3, which is adapted from a bioinformatics motif search program [16]. In this problem, candidates in the input stream are separated by the control character 'y'. The computation should report the candidates within a Hamming distance of d from the string stored in variable s. We use an `either/orelse` statement to ensure that computation continues to the next candidate when the current candidate does not fall within the threshold. The first block of the `either/orelse` statement performs the Hamming distance comparison, while the second block consumes input until the control character is reached, always preparing the program to check the next candidate.

Some Statements. In certain cases, for example instantiating macros based on the content of an array, the ability to generate a dynamic number of parallel paths is desirable. The `some` statement provides this functionality.

This statement is the parallel dual of a `foreach` loop. During execution, the program iterates over a provided array or string and instantiates a parallel thread of execution for each item. Similar to an `either/orelse` statement, the execution of each parallel thread continues with the subsequent statement in the program; different threads in the some statement may reach this next statement at disjoint times. The `some` statement in Figure 1 instantiates a Hamming distance

```
1  whenever( ALL_INPUT == input() ) {
2      foreach(char c : "rapid")
3          c == input();
4      report;
5  }
```

Figure 4: Execution of a sliding window search over the entire input stream for the string "`rapid`"

(a)   (b)

Figure 5: A behaviorally equivalent NFA and homogeneous NFA (both accept exactly `aa`, `aab`, and `aaca`)

macro for each string in the `comparisons` array. The number of parallel threads executed depends on the number of entries in `comparisons`.

Whenever Statements. A common operation in pattern-matching algorithms is a *sliding window* search, in which a pattern could begin on any character within the input stream. The `whenever` statement consists of a boolean guard and an internal statement, which is often a block. The guard specifies a condition on the input stream that must be true or a counter threshold that must be met before the internal statement is executed. At any point in the data stream where this guard is satisfied, the internal statement will be executed in parallel with the rest of the program. A `whenever` statement is the parallel dual of a while statement. Whereas a while statement checks the guard condition before each iteration of the internal statement, a `whenever` statement checks the guard in parallel with all other computations, if any.

The code fragment in Figure 4 will perform a sliding window search for the string "`rapid`". The predicate within the guard will return true on any input, and therefore the block of code will begin execution at every character in the input stream. The `whenever` statement can also perform restricted sliding window searches depending on the predicate in the guard. For example, an application searching through HTTP transaction might use the predicate matching "`GET`" before matching specific URLs.

Sliding window searches are fundamental to stream pattern recognition. All RAPID programs perform a sliding window search on the `START_OF_INPUT` symbol. In the common case, this sliding window search occurs at the topmost level of a RAPID program, i.e. right within the network. To reduce verbosity, RAPID infers this `whenever` statement, only requiring developers to specify a whenever statement when performing a non-default sliding window search.

4. The Automata Processor

Before describing our techniques for transforming RAPID programs into NFAs for acceleration with the AP, we present an overview of the device's architecture and computational model.

The Automata Processor, as described by Dlugosch et al. [10], is a hierarchical, memory-derived architecture for direct execution of homogeneous non-deterministic finite automata. NFAs are a useful model of computation for identifying patterns in a string of symbols. An NFA, formally,
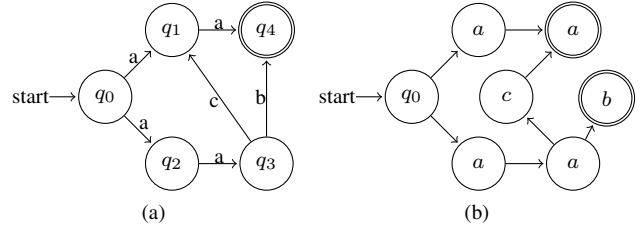
is defined as a tuple consisting of: a finite set of states, a finite alphabet, a transition function, the initial state, and the set of accepting states. The finite alphabet defines the allowable symbols within the input string. The transition function takes, as input, a set of currently active states and a symbol, and the function returns a new set of active states.

Traditionally, NFAs are often represented as a directed graph with states as vertices and the transition function encoded as edges. The AP executes an alternate form of NFAs known as *homogeneous* NFAs. These automata restrict the possible transition rules such that all incoming transitions to a state must occur on the same symbol.[1] Because all transitions to a state occur on the same symbol, we can label states with symbols rather than labeling the transitions. We refer to these combined states and labels as *state transition elements* (STEs). An STE *accepts* the symbols in its label, which we we refer to as the *character class* of the STE. Figure 5 depicts an NFA and a behaviorally equivalent homogeneous NFA. Additionally, the AP relaxes the definition of machine acceptance. Instead of accepting if an accepting state is active at the end of input, the AP will report any time an accepting state is active and record the relative offset in the input stream of this activation. This allows for pattern-recognition in streams of data symbols.

The set of languages recognized by both forms of finite automata are equivalent [6]; however, the homogeneous formulation allows for efficient execution directly in hardware. The STEs are stored in a memory array, and transitions between these STEs are embedded in a reconfigurable routing matrix. These two components—the memory array and routing matrix—form the basis of the AP architecture.

An SDRAM memory array serves as a computational medium in the AP, a stark contrast from its traditional role as main memory in a Von Neumann computer system. Arranged as a two dimensional grid, SDRAM data is accessed via row and column addresses. STEs consist of a single column of memory and a detection cell used for storing whether the given STE is active. The design in Figure 5b would re-

---

[1] NFAs traditionally support ϵ-transitions between a source and target state *without* consuming a symbol. These are not supported in our definition of homogeneous NFAs. An ϵ-transition may be removed by duplicating all incident transitions to the source state on the target state.
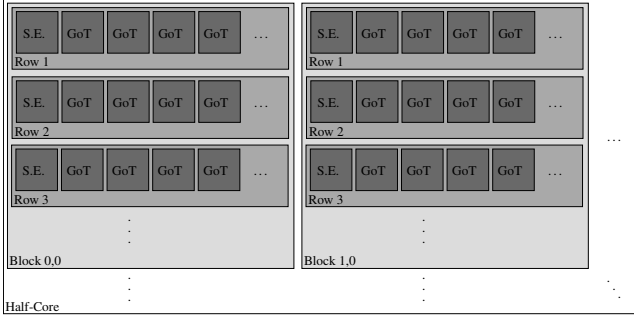
Figure 6: Hierarchical relationships between AP components: *groups of two* (GoT) and a *special purpose element* (S.E.) form a *row*, rows form *blocks*, and blocks form a *half-core*

Table 1: Resources available on the first-generation AP board, containing 32 chips

Total STEs	Total Counters	Total Boolean Logic Elements	Total Blocks	Half-Cores / Chip
1,572,864	24,576	73,728	6,144	2

STEs / Row	Rows / Block	Counters / Block	Boolean / Block	Blocks / Half-Core
16	16	4	12	96

quire seven columns of SDRAM, one for each of the STEs. The symbol or symbols accepted by an STE are stored in the column of memory, each row representing a different symbol in the alphabet. At runtime, a symbol from the input data stream is decoded and drives one of the rows in the memory array. Simultaneously, all STEs (columns of memory) determine whether they accept that symbol. Accepting STEs (i.e., those currently active as determined via their detection cells) then generate an output signal that is passed through the routing matrix to activate the connected STEs.

In addition to STEs, there may be additional special purpose elements. For example, the current generation AP contains saturating counters and combinatorial logic. These elements connect to the STEs via transition edges and allow for aggregation and thresholding of transitions between STEs. While these elements do not necessarily add any expressive power over traditional NFAs, the use of counters and logic often reduces the overall size of automata. This allows the AP architecture to be flexible. Future implementations might contain additional special purpose elements.

A hierarchical, reconfigurable routing matrix is used to route activation signals between STEs and special elements. Groupings of two STEs form a *GoT*. Several GoTs and a special purpose element (S.E.) are then connected to form a *row* in the routing matrix. Groupings of rows form *blocks*, and groupings of blocks form a *half-core*. Although an AP *chip* consists of two half-cores, there exists no routing between these two units. An AP *board* consists of several AP chips, and this allows for many pattern-recognition searches to occur in parallel. Figure 6 depicts this hierarchy for a half-core. Table 1 provides resource information for the first-generation AP board. We can take advantage of this hierarchical routing to produce efficient automata from RAPID programs.

5. Code Generation and Staged Computation

In this section, we present techniques for converting RAPID programs into automata for execution on the Automata Processor. The placement and routing tools for the AP accept both regular expressions and an XML-based design language called ANML as input. We choose to generate ANML because this allows for finer-grained control over the resulting automata. Additionally, ANML allows us to explicitly take advantage of future extension to the AP hardware, such as the addition of new special purpose elements.

Our technique takes two files as input: the RAPID program and a file annotating properties of the arguments to the network parameters, such as lengths of arrays and strings. Using this information, our tool converts the RAPID program into two files: an ANML specification and host driver code. The ANML file specifies the configuration of the AP needed to perform the given pattern-matching algorithm given by the RAPID program. The driver code is executed on the CPU at runtime to load the ANML design onto the AP, stream input data to the AP, and collect report events back from the AP. In this section, we focus on the transformation of RAPID into the ANML specification.

We employ a staged computation model to convert RAPID program: comparisons with the input stream and counters occur at runtime, while all other values are resolved at compile time. To aid in partitioning, we annotate all expressions with their return type during type checking. Allowable annotations include the five types listed in Section 3.2 as well as an internal `Automata` type, which we use to denote expressions interacting with the input stream. Expressions annotated with `Automata` or `Counter` are converted into ANML structures (allowing for runtime execution), while the remaining expressions are evaluated during compilation.

At its heart, our conversion algorithm recursively transforms RAPID programs into finite automata in much the same way that regular expressions can be recursively transformed into NFAs. Comparisons with the input stream are transformed into STEs. The particular statement in which the comparison occurs determines how the STEs attach to the rest of the automaton. Rules for transforming automata expressions determine the structure of the STEs within a given statement. We describe the conversion of expressions, statements and counters in turn.
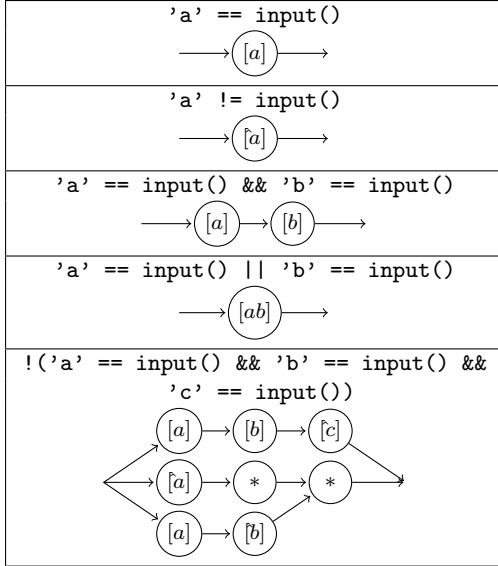
Figure 7: Example transformations of RAPID expressions into automata

(a) Foreach Loops

(b) Either/Orelse and Some statements

(c) While Loops

(d) Whenever statement

Figure 8: Automaton designs for RAPID statements

5.1 Converting Expressions

Expression transformation results in the formation of a chain of STEs. No cycles are generated by expressions, but chains may include bifurcations. Figure 7 provides examples of transformations from RAPID expressions to automata structures. The most basic transformation is a comparison between a character and the input stream, which generates a single STE. AND expressions behave as concatenation because reading from the input stream is destructive. The conversion of an OR expression generates a bifurcation in the generated automaton structure. A special case occurs when both sides of the OR expression contain input comparisons of length one. In these instances, we take advantage of STE character classes to specify multiple accepting symbols for a single STE.

Negations of expression generate the most complex structures of all the expression types. Traditionally, an automaton is negated by swapping accepting and non-accepting states. This construction, however, does not work for our use case because RAPID programs consume the same number of symbols for an expression and its negation. The traditional transformation does not maintain this property. Instead, we transform the expression via De Morgan's laws and generate STEs for the resulting statement. After any mismatch in this negation, the remaining symbols do not matter, but still must be consumed. We therefore use *star* states, which match on any character. In practice, complex STE character classes can handle such reserved symbols efficiently.

5.2 Converting Statements

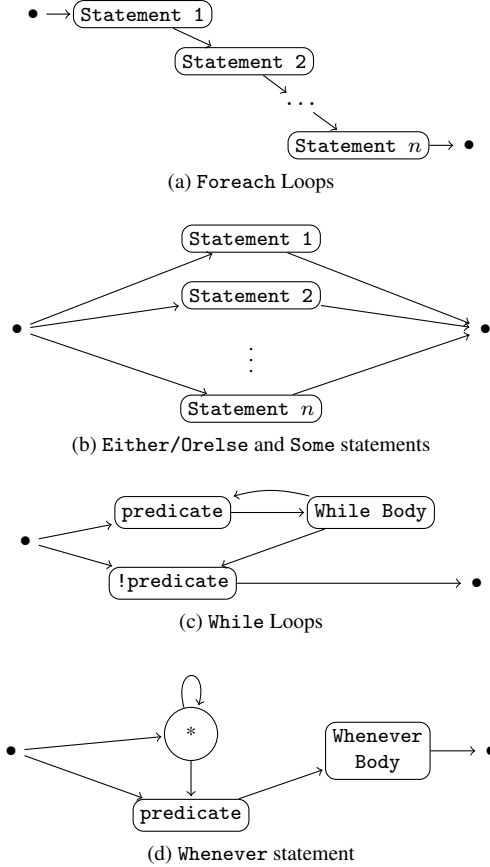Statements in RAPID are transformed into the high-level automaton structures, allowing for additional pipelining, feedback loops, and parallel exploration of patterns. We present the overall structures in Figure 8.

A foreach loop is unrolled into straight-line pattern-matching. Parallel either/orelse and some statement are transformed by generating the code for each statement and connecting these structures in parallel into the overall design. This mirrors the language semantics that the some statement is the parallel dual of foreach. Note that some statements typically depend on compile-time parameters (via input annotations on the network) while either/orelse statements do not (see Section 3.3).

There is also a similarity between while loops and whenever statements. While loops are transformed to alternately perform predicate checks and execute the body code. This generates a feedback loop structure in the automaton. In a whenever statement, predicate checking begins on every character consumed. To support this, we generate a self-activating STE that accepts all symbols (see ∗ node in Figure 8d). This added STE maintains an active transition into the predicate, allowing matching to begin on every symbol consumed. Once the predicate accepts, the body of the whenever statement will begin to execute (although the predicate is still checked again in parallel on subsequent input characters).

Table 2: Rules for thresholds and outputs on counters

Comparison	Threshold	True Output
< x	x	inverted
<= x	x+1	inverted
> x	x+1	non-inverted
>= x	x	non-inverted
== x	convert to <= x && >= x	
!= x	convert to < x || > x	

Figure 9: Structure of `whenever` statement with counters

5.3 Converting Counters

Counters in RAPID programs are transformed into a combination of one or more physical saturating counters and boolean logic elements. The basic structure consists of a saturating counter set to latch (once the threshold is reached, the output signal remains active) and an inverter, which allows for detection of the counter target not being reached.

Physical counters on the AP have three connection ports: count enable, reset, and output. `Counter` object function calls to `count()` and `reset()` in RAPID are connected to their respective ports on the counter. Output signals then connect to the next statement in the program.

We follow a set of rules for determining the threshold and outputs of a `Counter` object shown in Table 2. Equality checking with a `Counter` requires the use of two physical counter elements in the AP. While traversing the program, we note which `Counter` objects are used for equality checking. Then during code generation, we emit two counter elements for each.

This technique only allows for one threshold to be checked per counter in the RAPID program. An alternate solution would be to use positional encodings. This technique duplicates the automaton for each value the counter might have, encoding the count in the position of states within the automaton. While this design allows for easy checking of multiple thresholds, it also significantly increases the number of states in the final automaton and does not support counter resetting. We chose not to implement this technique in our initial compiler because it does not support full, generic functionality.

We must also support the use of `Counter` variables as predicates in a `whenever` statement. For the body of a `whenever` statement to execute, the `Counter` must have reached its appropriate threshold, and the statement itself must have been reached within the control flow of the RAPID program. We use a self-activating STE matching all symbols to track when the statement is reached. We use an AND gate to check both of these conditions before executing the body of the `whenever` statement. This design is demonstrated in Figure 9.

`Counter` threshold checks are also used as assertions or as predicates in `if` statements and `while` loops. Because NFAs do not have dynamic memory (beyond the states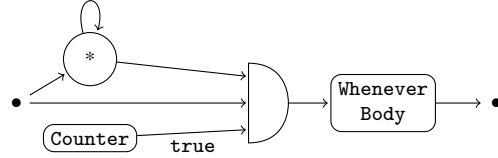 themselves), we propose to handle this case by both generating automata and also pre-transforming the input stream. For each such `Counter`, we create a unique reserved input symbol. This new symbol indicates that the threshold for that particular `Counter` has been met. We add an STE matching exactly that symbol linked to the subsequent statement; whenever that symbol is encountered in the input data stream, the appropriate subsequent statement begins execution. This symbol must be injected into the input data stream before the RAPID program begins execution. Actual injection is handled by the runtime code and can occur while data is being streamed to the AP (but before execution of the RAPID program beings).

We attempt to automatically determine the pattern for inserting the count threshold symbol into the input stream. An example pattern is "insert the symbol after every 25 characters in the input stream." In many cases, the compiler can infer the pattern by counting the number of symbols consumed before the counter check occurs. When `while` loops are included in the program, however, it may not be possible to determine where in the input stream to inject the unique symbols. In these cases, we currently output a warning at compile time and rely on the developer to provide the pattern for inserting the control character into the data stream.

6. Auto-tuning Tessellation

The automata designs generated from RAPID programs are often repetitive. Specifically, we found four out of five of our benchmarks reuse the same automaton structure multiple times, only altering the symbol sets of the STEs. Developers may take advantage of this property of designs to improve placement and routing times for the Automata Processor.

Using current tools for the AP, an automaton can be pre-compiled (i.e., placement and routing can be determined for a small portion of the overall design). State symbols are parameterized, allowing pre-compiled designs to be used repeatedly with different symbols. The developer creates automata that reference this pre-compiled design many times. This design is then placed and routed to be loaded onto the device. Because the individual automata have already been placed, the time needed is significantly reduced compared to processing the entire design at once; however, this process still requires over an hour for designs spanning an entire AP board.

We propose an alternate approach, *auto-tuning tessellation*, that requires seconds to produce final placement and

routing data. This optimization approximates filling an entire AP board with a design. Instead of generating a design to fill the entire board, we produce a design on a block-level for the AP (see Section 4). At runtime, we load this design several times onto the AP to fill the entire board. To improve device utilization (the number of automata loaded onto the AP), we attempt to generate dense designs at the block level.

To implement this optimization, we must determine which program elements to generate for tessellation. One option would be to require developers to annotate programs, indicating code sections to tile. This, however, places undue burden upon developers as they would have to analyze programs to determine the best locations to introduce tiling. Instead, we use a heuristic: code iterating over a network parameter within a some statement contained at the top level of the network is tiled. This heuristic captures the intended optimization behavior for all of our benchmarks that support tiling.

Given a portion of the code to tile, we generate an automaton. This design is placed and routed, reporting the block-level device utilization. Then, we iteratively add additional copies of the automaton to the design until just before device utilization increases. This technique improves overall usage by embedding more automata into each block.

7. Evaluation

We evaluate RAPID against hand-crafted designs for a collection of five benchmark applications, which were selected based upon previous research demonstrating significant acceleration using the Automata Processor [4, 16, 19, 21].

Table 3 provides descriptions of the benchmarks used. For each benchmark, we chose an instance size representative of a real-world problem. These sizes come either directly from previous work or from conversations with the authors of the previous work. The generation method column indicates the technique used to create the handcrafted code, which ranged from custom Java or Python programs for generating an ANML design to the use of a GUI design tool (Workbench) for crafting automata by hand. The authors of the *ARM* [19] and *Brill* [21] benchmarks provided us with their original code, including a collection of regular expressions for performing the *Brill* benchmark. We recreated the remaining designs, using algorithms and specifications published in previous work.

Table 4 lists design statistics for each of the benchmarks. We compare the lines of code needed to generate ANML, be it using RAPID or a custom technique in the cases of *ARM* and *Brill*. For *ARM*, the RAPID code requires six times fewer lines to represent, and *Brill* requires about half of the lines of the hand-crafted solution. The regular expression representation for *Brill* is more compact than RAPID. An author from the previous work [21] noted, however, that she found developing the regular expressions to be tedious and error-prone and felt that the RAPID program was more intuitive.

We created the *Gappy*, *Exact*, and *MOTOMATA* benchmarks using a GUI design tool. For these, we present the lines of code in ANML, which is roughly equivalent to the number of actions taken within the design tool. ANML file sizes are dependent on the specific instance of a problem. The numbers we present here are for a single instance of the problem size listed in Table 3. In all cases, the RAPID program is significantly more compact than the ANML it generates.

As an approximation for the size of the resulting automaton, we measure the number of STEs generated and the number of STEs loaded to the AP after placement and routing. The placement and routing tools modify the original automaton to better match the architectural design of the AP. For most benchmarks, RAPID-generated automata contain fewer device STEs, taking up less space on the device. Only the *Gappy* benchmark requires more device STEs. Although we could optimize the RAPID code to reduce the size of the generated automaton, we found that this more natural design, although larger, has comparable placement and routing efficiency. In the case of *MOTOMATA*, the RAPID version requires approximately half the STEs of the hand-crafted version. The compiled RAPID version makes use of a saturating counter, while the handcrafted version uses positional encoding.

In Table 5, we present the performance of RAPID programs compared to hand-crafted ANML based on placement and routing statistics for the AP. We use version 1.4-11 of the AP SDK to generate the placement and routing information. The total blocks column measures the number of routing matrix blocks (see Section 4) needed to accommodate the design; lower numbers represent a more compact design. STE utilization indicates the percent of used STEs within the routed blocks; high numbers indicate a design with fewer unused STEs. Mean BR allocation is a metric provided by the AP SDK that approximates of the routing complexity of the design. Here, a lower number is better, signifying lower congestion within the routing matrix. The clock divisor indicates whether the clock cycle of the AP must be reduced to accommodate a design. In one instance (the RAPID *MOTOMATA* program), the clock cycle must be halved due to a limitation in signal propagation between counters and combinatorial elements in the current generation AP. However, the RAPID version is four times more compact. Although this is a performance loss for a single instance, it is a net performance gain for a full problem, which will fill the AP board: four times as many instances execute in parallel at half the speed, for a net improvement factor of two. Although RAPID provides a higher level of abstraction than ANML, the final device binaries are more compact, using fewer resources on the AP.

Table 3: Description of benchmarks

Benchmark	Description	Generation Method	Sample Instance Size
ARM [19]	Association rule mining	Python + ANML	24 Item-Set
Brill [21]	Rule re-writing for Brill part of speech tagging	Java	219 Rules
Exact [4]	Exact match DNA sequence search	Workbench	25 Base Pairs
Gappy [4]	DNA string search with allowances for gaps between characters	Workbench	25-bp, Gaps $\leqslant 3$
MOTOMATA [16]	Fuzzy matching for planted motif Search in bioinformatics	Workbench	(17,6) Motifs

Table 4: Comparison between RAPID and hand-crafted code with respect to lines of code (LOC) and STE usage

Benchmark		LOC	ANML LOC	STEs	Device STEs
ARM	R	18	214	58	56
	H	118	301	79	58
Brill	R	688	10,594	3,322	1,429
	H	1,292	9,698	3,073	1,514
	Re	218	–[‡]	4,075	1,501
Exact	R	14	85	29	27
	H	–[†]	193	28	27
Gappy	R	30	2,337	748	399
	H	–[†]	2,155	675	123
MOTOMATA	R	34	207	53	72
	H	–[†]	587	150	149

R – RAPID   H – Hand-coded   Re – Regular Expression
[†] The GUI-tool does not have a LOC equivalent metric.
[‡] No ANML statistics are provided by the regular expression compiler.

Table 5: Placement and routing statistics

Benchmark		Total Blocks	Clock Divisor	STE Util.	Mean BR Alloc.
ARM	R	1	1	21.9%	20.8%
	H	1	1	23.4%	20.8%
Brill	R	8	1	84.0%	52.6%
	H	12	1	57.9%	65.4%
	Re	10	1	71.4%	60.6%
Exact	R	1	1	10.9%	4.2%
	H	1	1	10.9%	4.2%
Gappy	R	2	1	89.5%	70.8%
	H	2	1	37.5%	77.1%
MOTOMATA	R	1	2	33.6%	75.0%
	H	4	1	17.2%	75.0%

R – RAPID   H – Hand-coded   Re – Regular Expression

Finally, we evaluate our tessellation optimization by generating designs filling the entire AP Board. The *Brill* benchmark is fixed in size and is not applicable for this optimization. For the remaining benchmarks, we compare generating the "full problem" using non-compiled ANML, pre-compiled designs, and RAPID's tessellation feature (see Section 6 for descriptions). The ANML is generated using a Python script and bindings for the AP tool-chain. We choose problem sizes for each benchmark to ensure that the worst

performing technique would still fit on an AP board (6,144 blocks) in each case.

The results of this optimization are given in Table 6. We report the average of ten runs of both the ANML generation and placement and routing steps for each benchmark. We measured each configuration on the same machine using both version 1.4-11 and version 1.6-0 of the AP placement and routing tools, reporting the best result obtained. Neither version of the AP tools could successfully place and route the *ARM* baseline or the *Gappy* benchmark with pre-compiled designs.

To determine the total number of blocks for RAPID tessellation, we multiply the problem size by the number of blocks needed for the RAPID program (Table 4) and divide by the number of automata generated by the tessellation optimization. We use compiler output to determine the total number of blocks for the baseline and pre-compiled techniques.

Our auto-tuning tessellation technique outperforms both the baseline and pre-compiled designs in terms of time and space. We hypothesize that the improved block utilization results from routing complexities within the automata designs. When this complexity is present, the baseline technique cannot provide a more efficient layout than when using our tessellation. The *Exact* benchmark forms chains of STEs, which are simpler to route, and therefore the placement and routing tools are able to overlap some of the automata across blocks in the routing matrix using the baseline technique. Additionally, our technique is up to four orders of magnitude faster during compilation, placement, and routing than the baseline and up to three orders of magnitude faster than using pre-compilation. We do not view the many-hour place-and-route times of the default approach as efficient in practice, especially when our tessellation reduces this time to seconds. Consequently, RAPID tessellation promotes rapid prototyping of designs. By significantly decreasing design testing time, tessellation can increase developer productivity and improve maintainability while maintaining efficient device utilization.

8.   Conclusions and Future Directions

This paper presents RAPID, a language for defining pattern-matching algorithms. RAPID is motivated by pattern-recognition processors, such as the Automata Processor, which greatly

Table 6: Tessellation optimization

Benchmark		Problem Size (# instances)	Total Blocks	Generate Time (sec)	Place and Route Time (sec)	Total Time (sec)
ARM	B[†]	8,500	–	5.38	–	–
	P	8,500	6,100	6.53	771.16	770.70
	R	8,500	2,125	3.70	0.41	4.12
Exact	B	46 000	4 730	24.52	22 011.10	22 035.62
	P	46 000	6 075	30.17	1 676.88	1 707.05
	R	46 000	5 750	0.80	0.08	0.88
Gappy	B	2,000	5,354	0.99	9158.00	9158.99
	P[†]	2,000	–	0.99	–	–
	R	2,000	4,000	9.17	2.20	11.36
MOTOMATA	B	1 500	5 320	1.49	5 874.51	5 875.99
	P	1 500	6 001	1.45	210.62	212.07
	R	1 500	1 500	2.26	0.37	2.63

B – Baseline (No Pre-Compilation)   P – Pre-Compiled Designs   R – RAPID Tessellation

[†] The current AP software is not able to support placement and routing for this benchmark.

accelerate pattern detection in streams of data, but lack easy-to-use programming models.

We present techniques for converting RAPID programs to designs that can be executed and accelerated on the Automata Processor. Although RAPID programs are written at a higher level of abstraction than current hand-crafted code, our evaluation indicates that RAPID programs have similar, if not better, device utilization. We plan to develop additional conversion techniques to support CPUs and other hardware architectures.

Additionally, we argue that our auto-tuning tessellation technique for generating pattern searches spanning the entire AP board increases developer productivity and program maintainability. Compile times are several orders of magnitude faster than current methods and produce device configurations that maintain high device utilization.

There many options for extending the RAPID language. For example, generated automata could be optimized to better support the placement and routing tools. The Automata Processor has specific constraints for routing between special elements and states. By optimizing RAPID programs for these constraints, we can improve device utilization and also reduce placement and routing times. Further, debugging tools for pattern-matching processor are limited. Tools aiding developers to generate short input sequences to test corner cases of their applications as well as techniques for debugging RAPID would further increase productivity and maintainability.

In conclusion, RAPID raises the level of abstraction for programming pattern-recognition processors, resulting in clear, concise, maintainable, and efficient programs. We develop a notion of macros and networks, which we argue improve program maintainability. The same macro may be called with differing arguments to generate automata for varying instances of a pattern-matching problem. This program structure maps well to both pattern-recognition problems and also the underlying pattern-recognition hardware. Additionally, RAPID provides parallel control structures to support common tasks in pattern-matching algorithms, such as sliding window searches. Compared to hand-crafted versions of pattern-matching algorithms, the RAPID definition is more concise, needing only a fraction of the lines to implement.

References

[1] R. Alur, P. Černý, P. Madhusudan, and W. Nam. Synthesis of interface specifications for Java classes. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 98–109, 2005.

[2] G. Ammons, D. Mandelin, R. Bodík, and J. R. Larus. Debugging temporal specifications with concept analysis. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 182–195, 2003.

[3] K. R. Apt, J. Brunekreef, V. Partington, and A. Schaerf. Alma-0: An imperative language that supports declarative programming. Technical report, 1997.

[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. `https://www.capgemini.com/resource-file-access/resource/pdf/big\_fast\_data\_the\_rise\_of\_insight-driven\_business-report.pdf`, 2015.

[6] P. Caron and D. Ziadi. Characterization of Glushkov automata. *Theoretical Computer Science*, 233(1):75–90, 2000.

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

[8] C. S. Corporation. Big data universe beginning to explode. `http://www.csc.com/insights/flxwd/78931-big\`

`_data\_universe\_beginning\_to\_explode`, 2012.

[9] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, Aug. 1975.

[10] P. Dlugosch, D. Brown, P. Glendenning, M. Leventhal, and H. Noyes. An efficient and scalable semiconductor architecture for parallel automata processing. *IEEE Transactions on Parallel and Distributed Systems*, 25(12):3088–3098, Dec. 2014.

[11] A. Halaas. A systolic VLSI matrix for a family of fundamental searching problems. *Integration VLSI Journal*, 1(4):269–282, Dec. 1983.

[12] A. Halaas, B. Svingen, M. Nedland, P. Sætrom, O. Snøve, Jr., and O. R. Birkeland. A recursive MISD architecture for pattern matching. *IEEE Transactions on Very Large Scale Integrated Systems*, 12(7):727–734, July 2004.

[13] A. Krishna, T. Heil, N. Lindberg, F. Toussi, and S. Vander-Wiel. Hardware acceleration in the IBM PowerEN processor: Architecture and performance. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, pages 389–400, 2012.

[14] H. Lu, K. Zheng, B. Liu, X. Zhang, and Y. Liu. A memory-efficient parallel string matching architecture for high-speed intrusion detection. *IEEE Journal on Selected Areas in Communications*, 24(10):1793–1804, 2006.

[15] Micron. Calculating Hamming distance. `http://www.micronautomata.com/documentation/cookbook/c\_hamming\_distance.html`.

[16] I. Roy and S. Aluru. Finding motifs in biological sequences using the Micron Automata Processor. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 415–424, May 2014.

[17] W. Thies, M. Karczmarek, and S. P. Amarasinghe. StreamIt: A language for streaming applications. *Proceedings of the 11th International Conference on Compiler Construction*, pages 179–196, Apr. 2002.

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

[19] K. Wang, M. Stan, and K. Skadron. Association rule mining with the Micron Automata Processor. In *29th IEEE International Parallel & Distributed processing Symposium*, 2015.

[20] H. Yamada, M. Hirata, H. Nagai, and K. Takahashi. A high-speed string-search engine. *IEEE Journal of Solid-State Circuits*, 22(5):829–834, 1987.

[21] K. Zhou, J. J. Fox, K. Wang, D. E. Brown, and K. Skadron. Brill tagging on the Micron Automata Processor. In *Proceedings of the 2015 IEEE 9th International Conference on Semantic Computing (IEEE ICSC 2015)*, pages 236–239, Feb. 2015.