RAPID Programming of Pattern-Recognition Processors

Kevin Angstadt  Westley Weimer  Kevin Skadron
Department of Computer Science
University of Virginia
{angstadt, weimer, skadron}@cs.virginia.edu
6. April 2016
Finding Needles in a Haystack

• Researchers and companies are collecting increasing amounts of data
• 44x data production in 2020 than in 2009
• Demand for real-time analysis of collected data
What is the common theme?

- Locate the most probable location for a DNA fragment in the human genome.
- Find products that are most commonly purchased together.
- Parse English text to identify historical records that are duplicates.
- Identify consumer sentiment based off of social media posts.
- Search for Higgs events based off on paths of subatomic particles.

Pattern Search Problems
Parallel searches

Key = Active Searches

... G G C T A T G G ...

Incoming Data

CGGCAT

ATCGA
Parallel searches

Key

Active Searches

Target Pattern

Incoming Data

CGGCGTCAT

ATCGA

CGGCAT
Parallel searches

Key

\[
\begin{align*}
&= \\
\text{Active Searches} & \rightarrow \text{Target Pattern} \\
\end{align*}
\]

\[
\text{Incoming Data}
\]

\[
\begin{array}{cccccc}
A & C & G & G & C & T & A & T \\
\end{array}
\]

\[
\begin{array}{ccc}
G & G & G \\
\end{array}
\]

\[
\begin{array}{ccc}
G & G & G \\
\end{array}
\]

\[
\begin{array}{ccc}
G & G & G \\
\end{array}
\]

\[
\begin{array}{ccc}
G & G & G \\
\end{array}
\]

\[
\begin{array}{ccc}
G & G & G \\
\end{array}
\]

\[
\begin{array}{ccc}
G & G & G \\
\end{array}
\]

\[
\begin{array}{ccc}
G & G & G \\
\end{array}
\]

\[
\begin{array}{ccc}
G & G & G \\
\end{array}
\]

\[
\begin{array}{ccc}
G & G & G \\
\end{array}
\]

\[
\begin{array}{ccc}
G & G & G \\
\end{array}
\]

\[
\begin{array}{ccc}
G & G & G \\
\end{array}
\]

\[
\begin{array}{ccc}
G & G & G \\
\end{array}
\]

\[
\begin{array}{ccc}
G & G & G \\
\end{array}
\]

\[
\begin{array}{ccc}
G & G & G \\
\end{array}
\]

\[
\begin{array}{ccc}
G & G & G \\
\end{array}
\]

\[
\begin{array}{ccc}
G & G & G \\
\end{array}
\]

\[
\begin{array}{ccc}
G & G & G \\
\end{array}
\]

\[
\begin{array}{ccc}
G & G & G \\
\end{array}
\]

\[
\begin{array}{ccc}
G & G & G \\
\end{array}
\]

\[
\begin{array}{ccc}
G & G & G \\
\end{array}
\]

\[
\begin{array}{ccc}
G & G & G \\
\end{array}
\]

\[
\begin{array}{ccc}
G & G & G \\
\end{array}
\]

\[
\begin{array}{ccc}
G & G & G \\
\end{array}
\]

\[
\begin{array}{ccc}
G & G & G \\
\end{array}
\]

\[
\begin{array}{ccc}
G & G & G \\
\end{array}
\]

\[
\begin{array}{ccc}
G & G & G \\
\end{array}
\]

\[
\begin{array}{ccc}
G & G & G \\
\end{array}
\]

\[
\begin{array}{ccc}
G & G & G \\
\end{array}
\]

\[
\begin{array}{ccc}
G & G & G \\
\end{array}
\]

\[
\begin{array}{ccc}
G & G & G \\
\end{array}
\]

\[
\begin{array}{ccc}
G & G & G \\
\end{array}
\]

\[
\begin{array}{ccc}
G & G & G \\
\end{array}
\]

\[
\begin{array}{ccc}
G & G & G \\
\end{array}
\]

\[
\begin{array}{ccc}
G & G & G \\
\end{array}
\]

\[
\begin{array}{ccc}
G & G & G \\
\end{array}
\]

\[
\begin{array}{ccc}
G & G & G \\
\end{array}
\]

\[
\begin{array}{ccc}
G & G & G \\
\end{array}
\]

\[
\begin{array}{ccc}
G & G & G \\
\end{array}
\]

\[
\begin{array}{ccc}
G & G & G \\
\end{array}
\]

\[
\begin{array}{ccc}
G & G & G \\
\end{array}
\]

\[
\begin{array}{ccc}
G & G & G \\
\end{array}
\]

\[
\begin{array}{ccc}
G & G & G \\
\end{array}
\]

\[
\begin{array}{ccc}
G & G & G \\
\end{array}
\]

\[
\begin{array}{ccc}
G & G & G \\
\end{array}
\]

\[
\begin{array}{ccc}
G & G & G \\
\end{array}
\]

\[
\begin{array}{ccc}
G & G & G \\
\end{array}
\]

\[
\begin{array}{ccc}
G & G & G \\
\end{array}
\]

\[
\begin{array}{ccc}
G & G & G \\
\end{array}
\]

\[
\begin{array}{ccc}
G & G & G \\
\end{array}
\]

\[
\begin{array}{ccc}
G & G & G \\
\end{array}
\]

\[
\begin{array}{ccc}
G & G & G \\
\end{array}
\]

\[
\begin{array}{ccc}
G & G & G \\
\end{array}
\]

\[
\begin{array}{ccc}
G & G & G \\
\end{array}
\]

\[
\begin{array}{ccc}
G & G & G \\
\end{array}
\]

\[
\begin{array}{ccc}
G & G & G \\
\end{array}
\]

\[
\begin{array}{ccc}
G & G & G \\
\end{array}
\]

\[
\begin{array}{ccc}
G & G & G \\
\end{array}
\]

\[
\begin{array}{ccc}
G & G & G \\
\end{array}
\]

\[
\begin{array}{ccc}
G & G & G \\
\end{array}
\]

\[
\begin{array}{ccc}
G & G & G \\
\end{array}
\]

\[
\begin{array}{ccc}
G & G & G \\
\end{array}
\]

\[
\begin{array}{ccc}
G & G & G \\
\end{array}
\]

\[
\begin{array}{ccc}
G & G & G \\
\end{array}
\]

\[
\begin{array}{ccc}
G & G & G \\
\end{array}
\]

\[
\begin{array}{ccc}
G & G & G \\
\end{array}
\]

\[
\begin{array}{ccc}
G & G & G \\
\end{array}
\]

\[
\begin{array}{ccc}
G & G & G \\
\end{array}
\]
Parallel searches

Key

\[ \text{Active Searches} = \text{Target Pattern} \]

Incoming Data

ATCGA

CGGCAT
Parallel searches

Key

Active Searches

Target Pattern

ATCGA

Incoming Data

CGGCAT

Active Searches = Target Pattern
Parallel searches

Key

Active Searches

Target Pattern

Incoming Data

C A T A C G G C T

T

T

T

T

T

CGG

CGG

ATCGA

CGGCAT
Parallel searches

Key

Active Searches

Target Pattern

Incoming Data

CGGCAT

ATCGA

…
Parallel searches

\[ \text{Key} \quad \text{Active Searches} = \text{Target Pattern} \]

Incoming Data

\[ \ldots \quad A \ C \ C \ C \ A \ T \ A \ C \ G \ G \ C \ G \ C \ T \ A \ G \quad \ldots \]

\[ \text{ATCGA} \]

\[ \text{CGGCAT} \]
Parallel searches

Key

Active Searches

Target Pattern

Incoming Data

ATCGA

CGG

G

G

CTA

…

G

A

C

C

G

A

T

A

C

G

G

C

T

A

…
Parallel searches

Key

= Active Searches

Target Pattern

Incoming Data

... T G A C C A T A C G G C T ...

... C G G G C T ...

ATCGA

CGGCAT

Active Searches

Target Pattern
Parallel searches

Incoming Data

Key

Active Searches

Target Pattern

= Target Pattern

CGGCA

ATCGA

CGGCA

Incoming Data
Parallel searches

Key

Active Searches

Target Pattern

Incoming Data

... G C T G A C C A T G G C T ...
Parallel Searches: Goals

- Fast processing
- Concise, maintainable representation
- Efficient compilation
  - High throughput
  - Low compilation time

Specialized Hardware

RAPID Programming Language
A researcher should spend his or her time designing an algorithm to find the important data, not building a machine that will obey said algorithm.
RAPID Programming

• Pattern-Based Data Analysis
• Automata Processor
• Current Programming Models
• RAPID Language Overview
• AP Code Generation and Optimizations
• Evaluation
Micron’s Automata Processor

- Accelerates identification of patterns in input data stream using massive parallelism
- Hardware implementation of non-deterministic finite automata
- 1 gbps data processing
- MISD architecture
Micron’s Automata Processor

• Implements homogeneous NFAs
  – All incoming edges to state have same symbol(s)
  – State Transition Element (STE)

• Memory-derived architecture
  – Memory as a computational medium
  – State consists of a column in DRAM array
  – Connections made with reconfigurable routing matrix partitioned into blocks

• 1.5 million states on development board
• Saturating Up Counter, Boolean Logic
Micron’s Automata Processor
Programming Workflow

Front End Language

Synthesis, Placement, and Routing

Compiled Binary

AP D480 Hardware

Source: www.micronautomata.com
Current Programming Models

**ANML**
- Automata Network Markup Language
- Directly specify homogeneous NFA design
- High-level programming language bindings for generation

**RegEx**
- Support for a list of regular expressions
- Support for PCRE modifiers
- Compiled directly to binary
Programming Challenges

• ANML development akin to assembly programming
  – Requires knowledge of automata theory and hardware properties
  – Tedious and error-prone development process
• Regular expressions challenging to implement
  – Often exhaustive enumerations
  – Similarly error-prone
Programming Challenges

• Implement **single instance** of a problem
  – Each instance of a problem requires a brand new design
  – Need for meta-programs to generate final design

• Current programming models place unnecessary burden on developer
RAPID Programming

• Pattern-Based Data Analysis
• Automata Processor
• Current Programming Models
• RAPID Language Overview
• AP Code Generation and Optimizations
• Evaluation
RAPID at a Glance

• Provides concise, maintainable, and efficient representations for pattern-identification algorithms
• Conventional, C-style language with domain-specific parallel control structures
• Excels in applications where patterns are best represented as a combination of text and computation
• Compilation strategy balances synthesis time with device utilization
Program Structure

• **Macro**
  – Basic unit of computation
  – Sequential control flow
  – Boolean expressions as statements for terminating threads of computation

• **Network**
  – High-level pattern matching
  – Parallel control flow
  – Parameters to set run-time values
Program Structure

Network

Macros
Program Structure

```
network ( ... ) {
  ...
}
macro qux (...) {
  ...
}
macro bar (...) { ... }
macro baz (...) { ... }
macro foo (...) { ... }
```
Data in RAPID

- Input data stream as special function
  - Stream of characters
  - `input()`
    - Calls to `input()` are synchronized across all active macros
    - All active macros receive the same input character
Counting and Reporting

- **Counter**: Abstract representation of saturating up counters
  - Count and Reset operations
  - Can compare against threshold
- **RAPID programs can report**
  - Triggers creation of report event
  - Captures offset of input stream and current macro
Parallel Control Structures

• Concise specification of multiple, simultaneous comparisons against a single data stream
• Support MISD computational model
• Static and dynamic thread spawning for massive parallelism support
• Explicit support for sliding window computations
Parallel Control Structures

<table>
<thead>
<tr>
<th>Sequential Structure</th>
<th>Parallel Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>if…else</td>
<td>either…orelse</td>
</tr>
<tr>
<td>foreach</td>
<td>some</td>
</tr>
<tr>
<td>while</td>
<td>whenever</td>
</tr>
</tbody>
</table>
Either/Orelse Statements

either {
    hamming_distance(s,d);  // hamming distance
    ’y’ == input();         // next input is ’y’
    report;                 // report candidate
} orelse {
    while(’y’ != input());  // consume until ’y’
}

• Perform parallel exploration of input data
• Static number of parallel operations
Some Statements

network (String[] comparisons) {
    some(String s : comparisons)
        hamming_distance(s,5);
}

• Parallel exploration may depend on candidate patterns
• Iterates over items, dynamically spawn computation
Whenever Statements

whenever( ALL_INPUT == input() ) {
    foreach(char c : "rapid")
        c == input();
    report;
}

• Body triggered whenever guard becomes true
• ALL_INPUT: any symbol in the input stream
Example RAPID Program

Association Rule Mining
Identify items from a database that frequently occur together
Example RAPID Program

If all symbols in item set match, increment counter

Spawn parallel computation for each item set

Sliding window search calls `frequent` on every input

Trigger `report` if threshold reached

```java
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;
    }
}
```
RAPID Programming

- Pattern-Based Data Analysis
- Automata Processor
- Current Programming Models
- RAPID Language Overview
- AP Code Generation and Optimizations
- Evaluation
System Overview

- **Input**
  - RAPID Program
  - Annotations

- **Output**
  - Driver Code
  - AP Binary

- **RAPID Compiler**
  - ANML
  - apcompile
Code Generation

- Recursive transformation of RAPID program
  - Input Stream $\rightarrow$ STEs
  - Counters $\rightarrow$ 1 or more physical counter(s)
- Similar to RegEx $\rightarrow$ NFA transformation
Optimizing Compilation

- RAPID programs are often repetitive
- Extract repeated design, and compile once
- Load dynamically at runtime and set exact values (tessellation)
RAPID Programming

• Pattern-Based Data Analysis
• Automata Processor
• Current Programming Models
• RAPID Language Overview
• AP Code Generation and Optimizations
• Evaluation
Reminder: Goals

• Fast processing ✓
• Concise, maintainable representation
• Efficient compilation
  – High throughput
  – Low compilation time
# Description of Benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
<th>Generation Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARM</td>
<td>Association Rule Mining</td>
<td>Meta Program</td>
</tr>
<tr>
<td>Brill</td>
<td>Brill Part of Speech Tagging</td>
<td>Meta Program</td>
</tr>
<tr>
<td>Exact</td>
<td>Exact DNA Alignment</td>
<td>ANML</td>
</tr>
<tr>
<td>Gappy</td>
<td>DNA Alignment with Gaps</td>
<td>ANML</td>
</tr>
<tr>
<td>MOTOMATA</td>
<td>Planted Motif Search</td>
<td>ANML</td>
</tr>
</tbody>
</table>
RAPID Lines of Code

- ARM
- Brill
- Exact
- Gappy
- MOTOMATA

Lines of Code:
- Handcrafted
- RAPID
RAPID is Maintainable

Task: Convert Hamming distance comparison of length 5 to length 12
Parallel searches

Maximize number of parallel active searches by reducing STE usage
Compilation Time

- ARM*
- Brill
- Exact*
- Gappy*
- MOTOMATA*

- Handcrafted
- RAPID

* RAPID Tessellation
Conclusions

• RAPID is a high-level language for **pattern-search algorithms**
• Three domain-specific **parallel control structures**, and **suitable data representations**
• Accelerate using the Automata Processor
• RAPID programs are **concise, maintainable, and efficient**
ADDITIONAL SLIDES
Expression Generation

\[ 'a' == \text{input()} \]
\[ \rightarrow [a] \]

\[ 'a' != \text{input()} \]
\[ \rightarrow [^a] \]

\[ 'a' == \text{input()} && 'b' == \text{input()} \]
\[ \rightarrow [a] \rightarrow [b] \]

\[ 'a' == \text{input()} || 'b' == \text{input()} \]
\[ \rightarrow [ab] \]

\[ !('a' == \text{input()} && 'b' == \text{input()} && 'c' == \text{input()}) \]
\[ \rightarrow [^a] \rightarrow [^b] \rightarrow [^c] \]
Code Generation

(a) Foreach Loops

(b) Either/Orelse and Some statements
Code Generation

(c) While Loops

(d) Whenever statement
Counter Generation

<table>
<thead>
<tr>
<th>Comparison</th>
<th>Threshold</th>
<th>True Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; x</td>
<td>x</td>
<td>inverted</td>
</tr>
<tr>
<td>&lt;= x</td>
<td>x+1</td>
<td>inverted</td>
</tr>
<tr>
<td>&gt; x</td>
<td>x+1</td>
<td>non-inverted</td>
</tr>
<tr>
<td>&gt;= x</td>
<td>x</td>
<td>non-inverted</td>
</tr>
<tr>
<td>== x</td>
<td>convert to &lt;= x &amp;&amp; &gt;= x</td>
<td></td>
</tr>
<tr>
<td>!= x</td>
<td>convert to &lt; x</td>
<td></td>
</tr>
</tbody>
</table>
Estimating Runtime

Given \( n \) finite automata each consuming \( m \) square blocks:

\[
\text{runtime} \approx \left[ \frac{n \cdot m}{6144} \right] \cdot 7.5 \times 10^{-3} \text{ seconds}
\]

(Does not include delay for filling report buffer!)
Executing Finite State Machines in DRAM

- Columns in DRAM store STE labels (Each STE is a single column)
- Reconfigurable routing matrix connects the STEs

**Input:**
Drives a Row

**Columns with “1”:**
STEs that accept input symbol

**Active States**

**Active States for Next Clock Cycle**
## RAPID LOC and STE Comparison

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>LOC ANML</th>
<th>LOC STEs</th>
<th>Device STEs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>R</td>
<td>H</td>
<td></td>
</tr>
<tr>
<td><strong>ARM</strong></td>
<td>18</td>
<td>214</td>
<td>58</td>
</tr>
<tr>
<td></td>
<td>118</td>
<td>301</td>
<td>79</td>
</tr>
<tr>
<td><strong>Brill</strong></td>
<td>688</td>
<td>10,594</td>
<td>3,322</td>
</tr>
<tr>
<td></td>
<td>1,292</td>
<td>9,698</td>
<td>3,073</td>
</tr>
<tr>
<td></td>
<td>218</td>
<td>‡‡</td>
<td>4,075</td>
</tr>
<tr>
<td><strong>Exact</strong></td>
<td>14</td>
<td>85</td>
<td>29</td>
</tr>
<tr>
<td></td>
<td>†‡</td>
<td>193</td>
<td>28</td>
</tr>
<tr>
<td><strong>Gappy</strong></td>
<td>30</td>
<td>2,337</td>
<td>748</td>
</tr>
<tr>
<td></td>
<td>†‡</td>
<td>2,155</td>
<td>675</td>
</tr>
<tr>
<td><strong>MOTOMATA</strong></td>
<td>34</td>
<td>207</td>
<td>53</td>
</tr>
<tr>
<td></td>
<td>†‡</td>
<td>587</td>
<td>150</td>
</tr>
</tbody>
</table>

*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.
## RAPID Compilation Statistics

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

*R – RAPID  H – Hand-coded  Re – Regular Expression*
## RAPID Compilation

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Problem Size (# instances)</th>
<th>Total Blocks</th>
<th>Generate Time (sec)</th>
<th>Place and Route Time (sec)</th>
<th>Total Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARM</td>
<td>B†</td>
<td>8,500</td>
<td>–</td>
<td>5.38</td>
<td>770.70</td>
</tr>
<tr>
<td></td>
<td>P</td>
<td>8,500</td>
<td>6,100</td>
<td>6.53</td>
<td>771.16</td>
</tr>
<tr>
<td></td>
<td>R</td>
<td>8,500</td>
<td>2,125</td>
<td>3.70</td>
<td>4.12</td>
</tr>
<tr>
<td>Exact</td>
<td>B</td>
<td>46,000</td>
<td>4,730</td>
<td>24.52</td>
<td>22,011.10</td>
</tr>
<tr>
<td></td>
<td>P</td>
<td>46,000</td>
<td>6,075</td>
<td>30.17</td>
<td>1,676.88</td>
</tr>
<tr>
<td></td>
<td>R</td>
<td>46,000</td>
<td>5,750</td>
<td>0.80</td>
<td>0.88</td>
</tr>
<tr>
<td>Gappy</td>
<td>B</td>
<td>2,000</td>
<td>5,354</td>
<td>0.99</td>
<td>9158.00</td>
</tr>
<tr>
<td></td>
<td>P†</td>
<td>2,000</td>
<td>–</td>
<td>0.99</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>R</td>
<td>2,000</td>
<td>4,000</td>
<td>9.17</td>
<td>2.20</td>
</tr>
<tr>
<td>MOTOMATA</td>
<td>B</td>
<td>1,500</td>
<td>5,320</td>
<td>1.49</td>
<td>5,874.51</td>
</tr>
<tr>
<td></td>
<td>P</td>
<td>1,500</td>
<td>6,001</td>
<td>1.45</td>
<td>210.62</td>
</tr>
<tr>
<td></td>
<td>R</td>
<td>1,500</td>
<td>1,500</td>
<td>2.26</td>
<td>0.37</td>
</tr>
</tbody>
</table>

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.