Accelerating Pattern Searches with Hardware

Kevin Angstadt
angstadt@virginia.edu
CS 6354: Graduate Architecture
7. April 2016
Who am I?

• Work with Wes Weimer and Kevin Skadron
• PL + Architecture
  – Programming models for non-traditional computation
  – Software resiliency in autonomous vehicles
• Interdisciplinary research is fun!
We’re producing lots of 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.
- Sniff TCP/IP packets to detect possible outside attacks.
- 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

Target Pattern

Incoming Data

\[ \begin{align*}
G & \quad C & \quad T & \quad G & \quad A & \quad C & \quad C & \quad A & \quad T & \quad A & \quad C & \quad G & \quad G \\
\end{align*} \]

\[ \begin{align*}
\text{CGGCAT} & \quad \text{ATCGA} \\
\end{align*} \]
How do we define these patterns?

RegEx
What is a **Finite Automaton**?

- A finite automaton consists of
  - An input alphabet, $\Sigma$
  - A set of states, $Q$
  - A starting state, $q_0$
  - A set of accepting states, $F \subseteq Q$
  - A transition function, $\text{transition}(\text{input}, q_i) = q_j$
Finite Automaton Example

Start

Check: 1110

Check: 1110...
Optimization: Non-determinism

• Can be in multiple states at one time
  – Can start in more than one state
  – Transition function returns a set of states
  – (ε-transitions: we’ll ignore these, but you’ll see them in literature)

• Why? More compact design!
  – Exponentially more compact than a (D)FA
What does this find?

Diagram:

Start → * → Dd → o → n → u → t

Start → u → g → h
Let's process a DFA/NFA.

<table>
<thead>
<tr>
<th></th>
<th>Processing Complexity</th>
<th>Storage Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>NFA</td>
<td>$O(n^2)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>DFA</td>
<td>$O(1)$</td>
<td>$O(\sum^n)$</td>
</tr>
</tbody>
</table>

Why is a CPU **bad** at doing this?

**Processing Bandwidth**

**Memory Bandwidth**
Architectural Underpinnings

VON NEUMANN BOTTLENECK

CPU

MEMORY
What we need

• Compact design of a NFA
• Ability to update all transitions in a single time step

We can achieve this by throwing away the conventional von Neumann architecture
Micron’s Automata Processor

- Altera Stratix IV GX230 FPGA
- JTAG header
- AS header
- UDIMM socket M2
- Reset button
- 2x4 Power Connector
- SODIMM socket M3_0
- SODIMM socket M3_1
- SODIMM socket M4_0
- SODIMM socket M4_1
- PCIe Gen2 x8 Connector
- 2GB DDR3 M1

Figure courtesy of Micron
Exploiting DRAM

Row Access results in **one** word being retrieved from memory
Using Bit-Parallelism with NFAs

• Bit-parallel closure: given a set of states, what is the set of reachable states?
• Symbol closure: given an input symbol, what states accept this symbol?
• Transition function: intersection (bitwise AND) of these two closures
Bit-Parallel Execution

Input Stream: 1011

- Bit-parallel closure(\{start\}):  
  - \{q_0, q_1, q_2\}
- Symbol closure(1):  
  - \{q_0, q_2, q_4\}
- Bit & Symbol:  
  - \{q_0, q_2\}
Bit-Parallel Execution

Input Stream: 1011

- Bit-parallel closure(\{q_0, q_2\}):  
  - \{q_0, q_1, q_2, q_4\}

- Symbol closure(0):
  - \{q_0, q_1, q_3\}

- Bit & Symbol:
  - \{q_0, q_1\}
Bit-Parallel Execution

Input Stream: 1011

- Bit-parallel closure({q₀, q₂}):
  - {q₀, q₁, q₂, q₃}
- Symbol closure(1):
  - {q₀, q₂, q₄}
- Bit & Symbol:
  - {q₀, q₂, q₄}
Input Stream: 1011

Bit-Parallel Execution

- Bit-parallel closure(\{q_0, q_2\}):
  - \{q_0, q_1, q_2, q_4\}
- Symbol closure(1):
  - \{q_0, q_2, q_4\}
- Bit & Symbol:
  - \{q_0, q_4\}

REPORT!
Architectural Design

• Memory Array (Symbol Closure)
  – 256 Rows (i.e. 256 Input Characters)
  – 49,152 Columns (i.e. 49,152 States)

• Routing Matrix (Bit-Parallel Closure)
  – Saturating Counters
  – Boolean Gates
The AP at a High Level

Row Access results in **49,152** match & route operations
Executing NFA in DRAM

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

**Input:** Drives a Row

<table>
<thead>
<tr>
<th>STE (0)</th>
<th>STE (1)</th>
<th>STE (2)</th>
<th>STE (3)</th>
<th>STE (4)</th>
<th>STE (5)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mb (2^1-3) (common)</td>
<td>Mb (2^1-3)</td>
<td>Mb (2^1-3)</td>
<td>Mb (2^1-3)</td>
<td>Mb (2^1-3)</td>
<td>Mb (2^1-3)</td>
</tr>
<tr>
<td>Mb (2^2-2) (common)</td>
<td>Mb (2^2-2)</td>
<td>Mb (2^2-2)</td>
<td>Mb (2^2-2)</td>
<td>Mb (2^2-2)</td>
<td>Mb (2^2-2)</td>
</tr>
<tr>
<td>Mb (2^1-3) (common)</td>
<td>Mb (2^1-3)</td>
<td>Mb (2^1-3)</td>
<td>Mb (2^1-3)</td>
<td>Mb (2^1-3)</td>
<td>Mb (2^1-3)</td>
</tr>
<tr>
<td>Mb (2^1-3) (common)</td>
<td>Mb (2^1-3)</td>
<td>Mb (2^1-3)</td>
<td>Mb (2^1-3)</td>
<td>Mb (2^1-3)</td>
<td>Mb (2^1-3)</td>
</tr>
<tr>
<td>Mb (2^1-3) (common)</td>
<td>Mb (2^1-3)</td>
<td>Mb (2^1-3)</td>
<td>Mb (2^1-3)</td>
<td>Mb (2^1-3)</td>
<td>Mb (2^1-3)</td>
</tr>
<tr>
<td>Mb (2^1-3) (common)</td>
<td>Mb (2^1-3)</td>
<td>Mb (2^1-3)</td>
<td>Mb (2^1-3)</td>
<td>Mb (2^1-3)</td>
<td>Mb (2^1-3)</td>
</tr>
</tbody>
</table>

Active States:

Columns with “1”:
STEs that accept input symbol

Active States for Next Clock Cycle
Why Hierarchical Routing?

<table>
<thead>
<tr>
<th>Half-Core</th>
<th>Half-Core</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Column</strong></td>
<td><strong>Column</strong></td>
</tr>
<tr>
<td><strong>Block</strong></td>
<td><strong>Block</strong></td>
</tr>
<tr>
<td><strong>Row</strong></td>
<td><strong>Row</strong></td>
</tr>
<tr>
<td>( \text{G} )</td>
<td>( \text{G} )</td>
</tr>
</tbody>
</table>

24
Group of Two

- Contain two State Machine Elements (SME)
  - Hardware embodiment of STE
  - Memory Cells
  - Detection Cell
- Common output
- Loopback and Chaining
Row

- Contain 8 GoTs
- 1 Special Element (Boolean or Counter)
- "Match Element" (Connected to two GoTs)
- Row Routing Lines
  - SMEs exhibit *parity* (can only connect to half of input routing lines)
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
7. April 2016
Programming Workflow

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
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.
Parallel Searches: Goals

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

Specialized Hardware

RAPID Programming Language
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

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

```
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

RAPID Compiler

ANML
apcompile

Driver Code

AP Binary

Output
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
Parallel searches

Maximize number of parallel active searches
Optimizing Compilation

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

Key = Active Searches

Maximize number of parallel active searches

Reduce STE usage!

ATCGA

CGGCAT
Generated STEs

![Bar chart showing the comparison of Handcrafted and RAPID methods across different STEs for ARM, Brill, Exact, Gappy, and MOTOMATA. The x-axis represents the number of STEs ranging from 0 to 1600, and the y-axis lists the methods.

- ARM: Handcrafted = 10, RAPID = 1400
- Brill: Handcrafted = 1500, RAPID = 1400
- Exact: Handcrafted = 10, RAPID = 100
- Gappy: Handcrafted = 200, RAPID = 400
- MOTOMATA: Handcrafted = 100, RAPID = 300]
Compilation Time

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

Time (seconds) 1000 10000

- 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