Designing and Presenting Programming Languages in the Broader Research Community

CS 6610
23. November 2015
Today’s Plan

**Part 1:** How are Programming Languages presented to diverse audiences?

**Part 2:** Create your own language in 8 easy steps! (aka an overview of the RAPID programming language)
Let’s Discuss the Readings

- Angstadt, Weimer, and Skadron’s *RAPID Programming of Pattern-Recognition Processors*, ASPLOS ’16 (to appear)
- Frigo, Leiserson, and Randall’s *The Implementation of the Cilk-5 Multithreaded Language*, PLDI ’98
## Case Study

<table>
<thead>
<tr>
<th></th>
<th>RAPID</th>
<th>Cilk-5</th>
<th>MacroLab</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Paper Audience</strong></td>
<td>PL/Arch/OS</td>
<td>PL</td>
<td>CPS</td>
</tr>
<tr>
<td><strong>Hardware Target</strong></td>
<td>Automata Processor (Pattern-Recognition Processors)</td>
<td>Shared Memory Multi-Processor</td>
<td>Wireless Sensor Networks</td>
</tr>
<tr>
<td><strong>Problem Addressed</strong></td>
<td>Abstraction</td>
<td>Abstraction</td>
<td>Abstraction</td>
</tr>
<tr>
<td><strong># New Features</strong></td>
<td>~7</td>
<td>~4</td>
<td>~3</td>
</tr>
<tr>
<td><strong>Optimization</strong></td>
<td>Tessellation</td>
<td>Fast/Slow Clone</td>
<td>DSCD</td>
</tr>
</tbody>
</table>
CIVL: The Concurrency Intermediate Verification Language

The framework includes a verification tool (Sec. 3), based on model checking and symbolic execution, which can verify the following standard properties: absence of assertion violations, deadlocks, memory leaks, improper pointer dereferences or arithmetic, out-of-bound array indexes, reads of uninitialized variables, and divisions by 0.

In addition to the standard properties, a large number of dialect-specific properties are verified. For MPI, these

Some CIVL features are of course found in programming languages. CIVL's $range$ and $domain$ types are borrowed used). Cilk [21] and Erlang [3] also provide spawn primitives, but don't allow nested procedures. One language which does provide all these features is Racket [20], and in fact the basic structure of a CIVL model can be represented in Racket in a straightforward way, though it is not clear how easily other aspects of C (e.g., pointers and heaps) could be represented.

The verifier can also produce a minimal counterexample when a property violation is found. This is an execution trace of minimal length culminating in failure—something which greatly facilitates understanding, isolating, and repairing defects. Again, this contrasts with most testing and debugging methods in HPC, which often involve traces with astronomical numbers of threads, processes, or execution steps.
Computer Scientists

- This Turing Award winner is best known for his work on distributed systems, including: notions of time, sequential consistency, the bakery algorithm for mutual exclusion, and the Byzantine Generals Problem. Additionally, he was an initial developer of LaTeX.
Trivia

• On November 9, 1989, government spokes-
person Günter Schabowski improvised an
answer to a reporter’s question ("As far as I
know, effective immediately … without
delay"), leading to a massive “upheaval” in
world politics. Name the event or the official
name of the country Shabowski represented.
RAPID Programming of Pattern-Recognition Processors
Understand your target (and your audience)

STEP 1
What is a Pattern?

• Sequence of data that should be found within another collection of data
• We’ll focus on NFAs/Regular Expressions
• Many problems can be phrased as pattern-recognition!
  – NLP: Brill tagging
  – ML: Association Rule Mining, Random Forests
  – Bioinformatics: Motif searches, K-mer matching
  – Entity Resolution
  – Network Intrusion Detection
What is the Automata Processor?

- DRAM-based engine for executing NFAs
- 1.57 million state capacity
- 133 MHz Clock Speed
- 213 trillion transitions per second
- Added 12-bit counters and Boolean logic
Homogeneous NFAs
Example: Build a Homogeneous NFA

.*[Dd](o|ough)nut
Row Access results in **one** word being retrieved from memory.

Row Access results in **49,152** match & route operations.
Current Programming Models

ANML  RegEx
ANML (Finite Automata)

from micronap.sdk import *

# Initialize the automaton
A = Anml()
AN = A.CreateAutomataNetwork(anmlId='hamming_distance')

# Add states
...
c = AN.AddSte('c', anmlId='c')
a = AN.AddSte('a', anmlId='a')
b = AN.AddSte('b', anmlId='b')
l = AN.AddSte('l', anmlId='l')
e = AN.AddSte('e', anmlId='e')
...

# Add transitions
...
AN.AddAnmlEdge(p1, a, 0)
AN.AddAnmlEdge(p2, b, 0)
AN.AddAnmlEdge(p3, l, 0)
AN.AddAnmlEdge(p4, e, 0)
...

# Export the automaton to an ANML file
AN.ExportAnml('hamming_distance.anml')

- Verbose (62 lines of code)
- Not easily maintained
- Conceptual burden
Regular Expressions

/ right/JJ to/[\s]+ /  
/ the/[\s]+ back/RB /  
/ [\s]/+DT longer/[\s]+ /  
/ ,/[\s]+ have/VB /  
/ [\s]/+VBD by/[\s]+ /  
/ her/PRP\$ ,/[\s]+ /  
/ [\s]/+DT right/[\s]+ /  
/ [\s]/+IN 's/[\s]+ /  
/ [\s]/+RBS of/[\s]+ /  
/ had/[\s]+ had/VBD /  
/ set/VBN of/[\s]+ /  
/ [\s]/+VBG room/[\s]+ /  
/ [\s]/+VB of/[\s]+ /  
/ [\s]/+JJ by/[\s]+ /  
/ one/CD 's/[\s]+ /

- Must know patterns at compile time
- Often exhaustive enumeration
  - How do we generate?
- Not easily maintained
Finding Motifs in Biological Sequences using the Microt Automata Processor

Indranil Roy and Srinivas Aluru
School of Computational Science and Engineering
Georgia Institute of Technology, Atlanta, GA 30332
Email: rtoy@gatech.edu; aluru@cc.gatech.edu

Abstract—Finding approximately conserved sequences, called motifs, across multiple DNA or protein sequences is an important problem of identifying one or more motifs which occur in all the sequences with up to |d| mismatches. The motif search problem has been traditionally solved by exact algorithms, but even these exact algorithms return the same answers, they are evaluated based on their execution time. For this evaluation, 20 sequences of length 600 are randomly generated by using characters from the alphabet. Similarly, a motif M of length |l| is generated and planted at a random location in each sequence. Run-times are only reported for instances of (|l|, |d|) which are known to be challenging. For these instances, the expected number of |M| motifs found in the sequences apart from the planted motif is at least 1. Examples of such instances are (25, 5), (19, 7), (21, 8), (23, 9), and (27, 10). At the time of this writing, the shortest execution times reported for the (20, 11) instance on a 48 core machine is 46.9 hours, and the (28, 12) instance remains unsolved to the best of our knowledge.

The Micron Automata Processor [21] is a novel non-Nios Numa processor which has been shown to be an interesting platform for developing novel algorithms for solving NP-complete hard problems. Our algorithms, termed MOTOMA-TA and MOTOMA, are both designed to efficiently search a large number of non-deterministic Finite Automata (NFA) in parallel. We estimate the run-time for the deployment that can simultaneously execute multiple NFAs in parallel. We propose a new accelerator technology for streaming execution over a large set of Non-deterministic Finite Automata (NFA). This system is designed to take advantage of the Micron Automata Processor, a new technology close to this new accelerator technology. In this paper, we serve as a useful introduction to the micromotif search problem using this new accelerator technology.

Index Terms—computational biology; finite automaton; graph algorithms; hardware acceleration; motif detection.

1. INTRODUCTION

Large repositories of genetic data have been produced through numerous sequencing projects, a trend that has sig-
nificantly accelerated during the last decade. An important part of the analysis of this data consists of finding patterns in DNA, RNA or protein sequences. Discovery of these patterns called motifs helps in the identification of transcription factor binding sites, transcriptional regulatory elements and their consequences on gene functions, viruses causing human diseases, and therapeutic drug targets.

The problem of motif search has been classified into the following three types: Packed motif search Problem (PMP), Simple motif search Problem (SMP), and k-allowance-based Motif search Problem (KMP). Of these three types, PMP has been studied in the prior work and described as follows: Given n sequences of length |l| each, find a motif M of length |l| which occurs in all the sequences with up to |d| mismatches. For example, given the input sequences ATGCT/CTGAC/TTACGCT/CGT and GCC/TAC/GGTGG/CTC and |d| = 1, the motif TAC occurs in all the three sequences. Note that the motif itself need not be present in its exact form in any of the sequences, which is the case in this example. The generic form of the PMP is defined as the Quorum Planted Motif search Problem (qPMP), where the motif M is present at least q of the n sequences. PMP (and therefore qPMP) has been shown to be NP-hard [1].

Though the first motif was discovered in 1970 [2], this problem is far from being satisfactorily solved. Heuristic algorithms [3], [4], [5], [6], [7] are currently being used to search for solutions faster, but are not guaranteed to identify all motifs [8], sometimes even equivalent ones. On the other hand, exact algorithms [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20] can identify all the motifs, but take a large amount of time and memory to do so. Since all exact algorithms return the same answers, they are evaluated based on their execution time. For this evaluation, 20 sequences of length 600 are randomly generated by using characters from the alphabet. Similarly, a motif M of length |l| is generated and planted at a random location in each sequence. Run-times are only reported for instances of (|l|, |d|) which are known to be challenging. For these instances, the expected number of |M| motifs found in the sequences apart from the planted motif is at least 1. Examples of such instances are (25, 5), (19, 7), (21, 8), (23, 9), and (27, 10). At the time of this writing, the shortest execution times reported for the (20, 11) instance on a 48 core machine is 46.9 hours, and the (28, 12) instance remains unsolved to the best of our knowledge.

The Micron Automata Processor [21] is a novel non-Nios Numa processor which can be pro-
grammed to execute thousands of Non-deterministic Finite Automata (NFA) in parallel to identify patterns in a data stream. We designed an exact algorithm for solving PMP and qPMP using NFA parallelism inside the Micron Automata Processor. Our algorithms, termed MOTOMA and MOTOMA, not only have lower execution times but can also handle more challenging instances of the problem which are inherently unsolved. To the best of our knowledge, this is the first instance of accelerating a complex application using the Automata Processor, and can be helpful to the community as this new accelerator technology becomes available.

The rest of the paper is organized as follows. A high-level description of the Micron Automata Processor is provided in Section II. Our parallel algorithm for the (|l|, |d|) motif search problem is described in Sections III through VI. Section VII contains the expected run-times for various sizes of the (|l|, |d|) motif search problem vis-a-vis prominent software based methods. Our findings are summarized in Section VIII.
Properties to Look For

• Well-described and defined problem with a known solution
• Not easily solved with current techniques
• Seems to be representative of a large class of problems in the application space
Hamming distance: the number differences in corresponding characters between two strings

Extract the KERNEL computation and implement
Think about code generation at the same time!

STEP 3
Pick a “Base” Language

• RAPID starts out with a C-like base language
• Add a few abstractions for ease of programming (foreach)
• Boolean expressions as statements (inspired by literature search)
• Make sure you can support the basic features of the architecture
Program Structure

• **Macro**
  – Basic unit of computation
  – Sequential control flow
  – Boolean Expressions as Statements
  – More dynamic than ANML macros

• **Network**
  – High-level pattern matching
  – Parallel control flow
  – Parameters to set run-time values
Types in Rapid

- int
- char
- boolean
- String (.length(), charAt(i), etc.)
- Counter (.count() and .reset())
- Arrays
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
Hamming Distance in RAPID

```
macro hamming_distance (String s, int d) {
    Counter cnt;
    foreach (char c : s)
        if(c != input()) cnt.count();
    cnt <= d;
    report;
}
```
Okay, that went well…
Let’s try to implement more of the algorithm (adding language features as we need them)

STEP 4
Need to Iterate Over Multiple Candidates in Patterns and Input Stream

• Special Symbols
  – ALL_INPUT
  – START_OF_INPUT

• We also need parallel control flow
Either/Orelse Statements

```java
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

• Parallel exploration may depend on candidate patterns
• We need a way of dynamically generating parallel computation
• Some statement is a “parallel for loop” over data
Some Statements

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 }
Association Rule Mining with the Micron Automata Processor

Ke Wang,* Yanjun Qi†, Jeffery J. Fox‡, Mineza R. Shaalan† and Kevin Skadron§

†Dept. of Comp. Sci., ‡Dept. of Mater. Sci., §Dept. of Elec. & Comp. Eng.
University of Virginia
Charlottesville, VA, 22904 USA
{kwang, yanjun, jff5x, mircea, skadron}@virginia.edu

Abstract—Association rule mining (ARM) is a widely used data mining technique for discovering sets of frequently occurring items in large databases. As databases grow in size and real-time analysis becomes important, the performance of ARM implementations can impede its applicability. We accelerate ARM by using Micron’s Automata Processor (AP), a hardware implementation of non-deterministic finite automata (NFAs), with additional features that significantly expand its capabilities beyond those of traditional NFAs. The AP implements the multipass pruning strategy used in the Apriori ARM through the AP’s symbol replacement capability, a form of lightweight reconfigurability. Up to 129X and 49X speedups are achieved by the AP-accelerated Apriori, as compared to single-core CPU implementation and Eclat, a more efficient ARM algorithm. 4-core multicore CPU implementations also suggest even better speedups from future generations of AP.

Keywords—Automata Processor; association rule mining; frequent set mining

I. INTRODUCTION

Association Rule Mining (ARM), also referred to as Frequent Set Mining (FSM), is a data mining technique that identifies strong and interesting relations between variables in databases using different measures of “interestingness”. ARM has been the key module of many recommendation systems and has created many commercial opportunities for on-line retail stores. In the past 10 years, this technique has also been widely used in web usage mining, traffic accident analysis, intrusion detection, market basket analysis, bioinformatics, etc.

As modern databases continue to grow rapidly, the execution efficiency of ARM becomes a bottleneck for its application in new domains. Many previous studies have been devoted to improving the performance of sequential CPU-based ARM implementations. Different data structures were proposed, including horizontal representation, vertical representation, and matrix representation [1]. Multiple algorithms have been developed including Apriori [2], Eclat [3] and FP-growth [4]. A number of parallel acceleration based solutions have also been implemented on multi-core CPU [5], GPU [6] and FPGA [7].

Recently, Micron proposed a novel and powerful non-von Neumann architecture, the Automata Processor (AP). The AP architecture demonstrates a massively parallel computing ability through a large number of state elements. It also achieves fine-grained communication ability through its configurable routing mechanism. These advantages make the AP suitable for pattern-matching centered tasks like ARM. Very recently, the AP has been successfully used to accelerate the tasks of regular expression matching [8] and DNA motif searching [9].

In this paper, we propose an AP-based acceleration solution for ARM. A Non-deterministic Finite Automata (NFA) is designed to recognize the sets of frequent items. Counter elements of the AP are used to count the frequencies of elements. We also introduce a number of optimization strategies to improve the performance of AP-based ARM. On multiple synthetic and real-world datasets, we compare the performance of the proposed AP-accelerated Apriori versus the Apriori single-core CPU implementation, as well as multicore and GPU implementations of Eclat algorithm.

The proposed solution achieves up to 129X speedups when compared with the Apriori single-core CPU implementation and up to 49X speedups over multicore implementation of Eclat. It also outperforms GPU implementations of Eclat in some cases, especially on large datasets. Overall, this paper makes three principal contributions:

1) We develop a CPU-AP computing infrastructure to improve the Apriori algorithm based ARM.
2) We design a novel automation structure for the matching and counting operations in ARM. This structure avoids routing reconfiguration of the AP during the mining process.
3) Our AP ARM solution shows performance improvement and broader capability over multi-core and GPU implementations of Eclat ARM on large datasets.

II. ASSOCIATION RULE MINING

Association rule mining (ARM) among sets of items was first described by Agrawal and Srikant [10]. The ARM problem was initially studied to find regularities in the

Iterate with new application examples
Do you need any additional program features?

STEP 5
Sliding Window Search

@NITBDELMVUDBQZZDWITHFPTG@ZBGEXDGHXSVCMKKQEQEYKPREBN...

Pattern

ARM needs custom sliding window!

All RAPID programs perform sliding window search on **START_OF_INPUT**
(automatically inferred)
Whenever Statements

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

- Parallel equivalent of while loop
- Body triggered whenever guard becomes true
STEP 6

Make sure you can generate code!
System Overview

Source-to-source

RAPID Program

Annotations

Input

RAPID Compiler

ANML

apcompile

Driver Code

Output

AP Binary
Program Structure

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

• Recursively transform RAPID program
  – Input Stream → States
  – Counters → 1 or more physical counter(s)
• Similar to RegEx → NFA transformation
Staged Computation

Runtime
Comparisons with Input Stream Counters

Compile Time
Integer Computation
Iteration Over Strings
etc.
Compiling for the full AP board

AUTO-TUNING TESSELLATION

Oh wait! It doesn’t compile well? That’s a pity…

STEP 7
Block-Level Configuration

Each computation takes up SPACE on the AP (TIME is dependent on the input)

Load blocks dynamically and set parameters at runtime
Auto-Tuning
Block-Level Configuration

- Easier to dynamically load designs at runtime
- Relatively fast compilation
- Auto-tuning heuristic improves utilization
Evaluation!

STEP 8
## Description of Benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
<th>Generation Method</th>
<th>Sample Instance Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARM</td>
<td>Association rule mining</td>
<td>Python + ANML</td>
<td>24 Item-Set</td>
</tr>
<tr>
<td>Brill</td>
<td>Rule re-writing for Brill part of speech tagging</td>
<td>Java</td>
<td>219 Rules</td>
</tr>
<tr>
<td>Exact</td>
<td>Exact match DNA sequence search</td>
<td>Workbench</td>
<td>25 Base Pairs</td>
</tr>
<tr>
<td>Gappy</td>
<td>DNA string search with allowances for gaps between characters</td>
<td>Workbench</td>
<td>25-bp, Gaps ≤ 3</td>
</tr>
<tr>
<td>MOTOMATA</td>
<td>Fuzzy matching for planted motif Search in bioinformatics</td>
<td>Workbench</td>
<td>(17,6) Motifs</td>
</tr>
</tbody>
</table>
## RAPID LOC Comparison

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>LOC</th>
<th>ANML LOC</th>
<th>STEs</th>
<th>Device STEs</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARM</td>
<td>R 18</td>
<td>214</td>
<td>58</td>
<td>56</td>
</tr>
<tr>
<td></td>
<td>H 118</td>
<td>301</td>
<td>79</td>
<td>58</td>
</tr>
<tr>
<td>Brill</td>
<td>R 688</td>
<td>10,594</td>
<td>3,322</td>
<td>1,429</td>
</tr>
<tr>
<td></td>
<td>H 1,292</td>
<td>9,698</td>
<td>3,073</td>
<td>1,514</td>
</tr>
<tr>
<td></td>
<td>Re 218</td>
<td>_†</td>
<td>4,075</td>
<td>1,501</td>
</tr>
<tr>
<td>Exact</td>
<td>R 14</td>
<td>85</td>
<td>29</td>
<td>27</td>
</tr>
<tr>
<td></td>
<td>H _†</td>
<td>193</td>
<td>28</td>
<td>27</td>
</tr>
<tr>
<td>Gappy</td>
<td>R 30</td>
<td>2,337</td>
<td>748</td>
<td>399</td>
</tr>
<tr>
<td></td>
<td>H _†</td>
<td>2,155</td>
<td>675</td>
<td>123</td>
</tr>
<tr>
<td>MOTOMATA</td>
<td>R 34</td>
<td>207</td>
<td>53</td>
<td>72</td>
</tr>
<tr>
<td></td>
<td>H _†</td>
<td>587</td>
<td>150</td>
<td>149</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.
# Device Utilization

<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>R 1</td>
<td>1</td>
<td>21.9%</td>
<td>20.8%</td>
</tr>
<tr>
<td></td>
<td>H 1</td>
<td>1</td>
<td>23.4%</td>
<td>20.8%</td>
</tr>
<tr>
<td><strong>Brill</strong></td>
<td>R 8</td>
<td>1</td>
<td>84.0%</td>
<td>52.6%</td>
</tr>
<tr>
<td></td>
<td>H 12</td>
<td>1</td>
<td>57.9%</td>
<td>65.4%</td>
</tr>
<tr>
<td></td>
<td>Re 10</td>
<td>1</td>
<td>71.4%</td>
<td>60.6%</td>
</tr>
<tr>
<td><strong>Exact</strong></td>
<td>R 1</td>
<td>1</td>
<td>10.9%</td>
<td>4.2%</td>
</tr>
<tr>
<td></td>
<td>H 1</td>
<td>1</td>
<td>10.9%</td>
<td>4.2%</td>
</tr>
<tr>
<td><strong>Gappy</strong></td>
<td>R 2</td>
<td>1</td>
<td>89.5%</td>
<td>70.8%</td>
</tr>
<tr>
<td></td>
<td>H 2</td>
<td>1</td>
<td>37.5%</td>
<td>77.1%</td>
</tr>
<tr>
<td><strong>MOTOMATA</strong></td>
<td>R 1</td>
<td>2</td>
<td>33.6%</td>
<td>75.0%</td>
</tr>
<tr>
<td></td>
<td>H 4</td>
<td>1</td>
<td>17.2%</td>
<td>75.0%</td>
</tr>
</tbody>
</table>

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

<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></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>B†</td>
<td>8,500</td>
<td>–</td>
<td>5.38</td>
<td>–</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>0.41</td>
</tr>
<tr>
<td>Exact</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></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>0.08</td>
</tr>
<tr>
<td>Gappy</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></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></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></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.
Conclusions

• RAPID admits clear, concise, maintainable, and efficient pattern-recognition programs
• Language development is an iterative process
• Make the common tasks easy (remember your target audience!)
• The fewer new features, the better (but they’ll have high impact!)