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?

Pattern Search Problems

Locate the most probable DNA fra… huma…

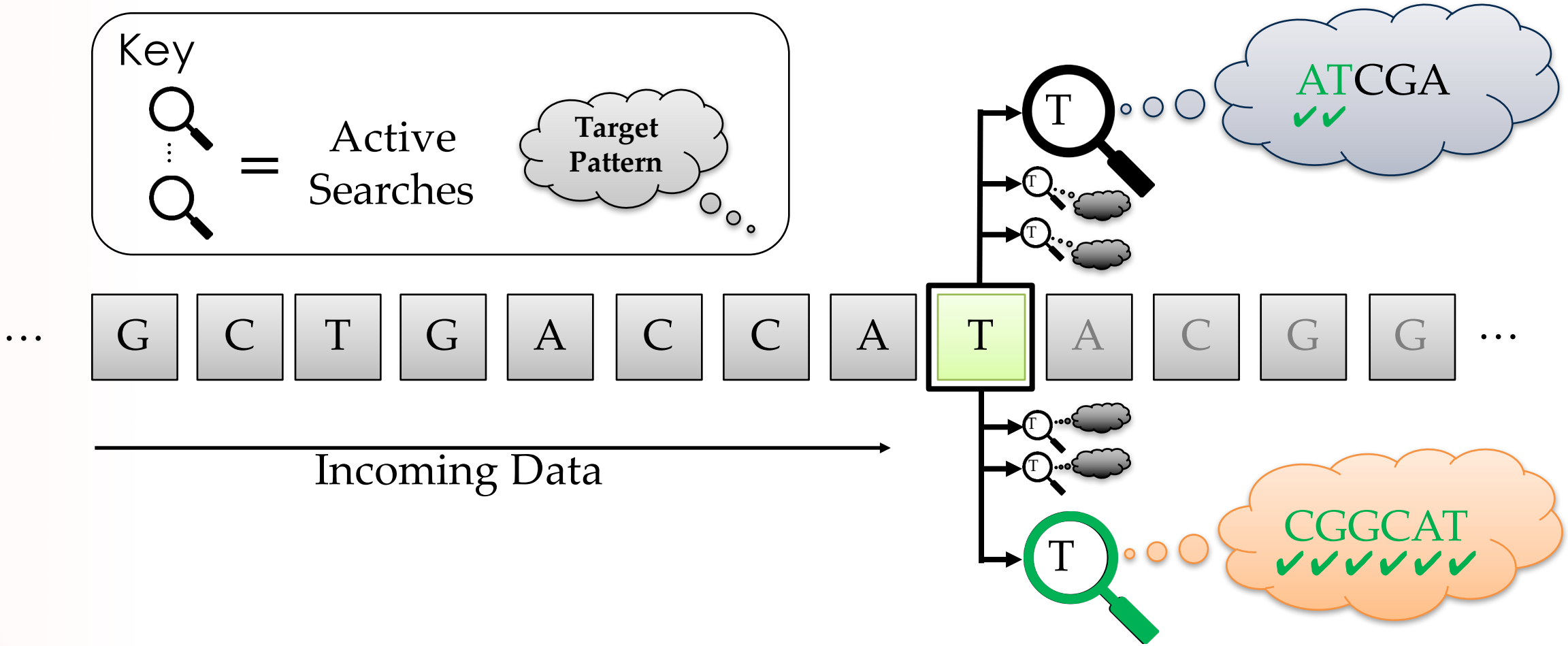
Find prod… most … purchas…

Sniff TCP/IP… detect po… a…

Identify … sentimen… social n…

Search for Higgs events based off on paths of subatomic particles

Parallel searches



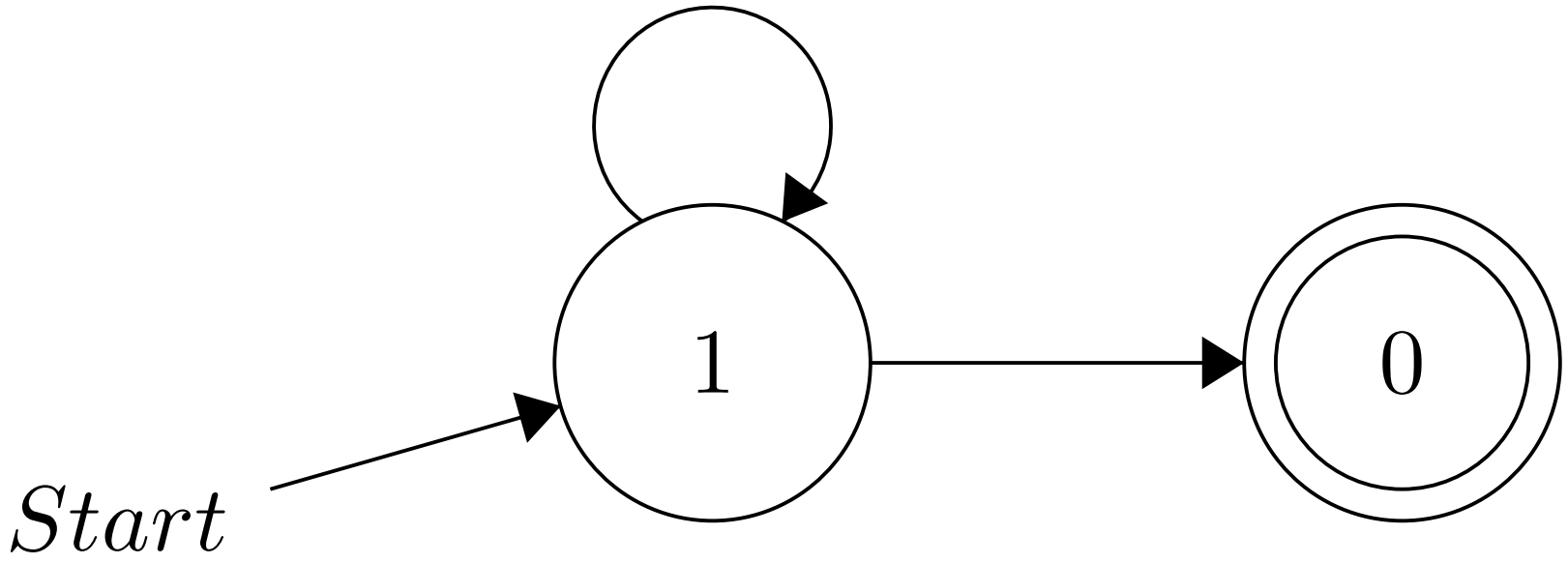
How do we define these patterns?

RegEx

What is a **Finite Automaton**?

- A finite automaton consists of
  - An input alphabet, Σ
  - A set of states, Q
  - A starting state, q_0
  - A set of accepting states, $F \subseteq Q$
  - A transition function, $transition(\ input, q_i\ ) = q_j$

Finite Automaton Example



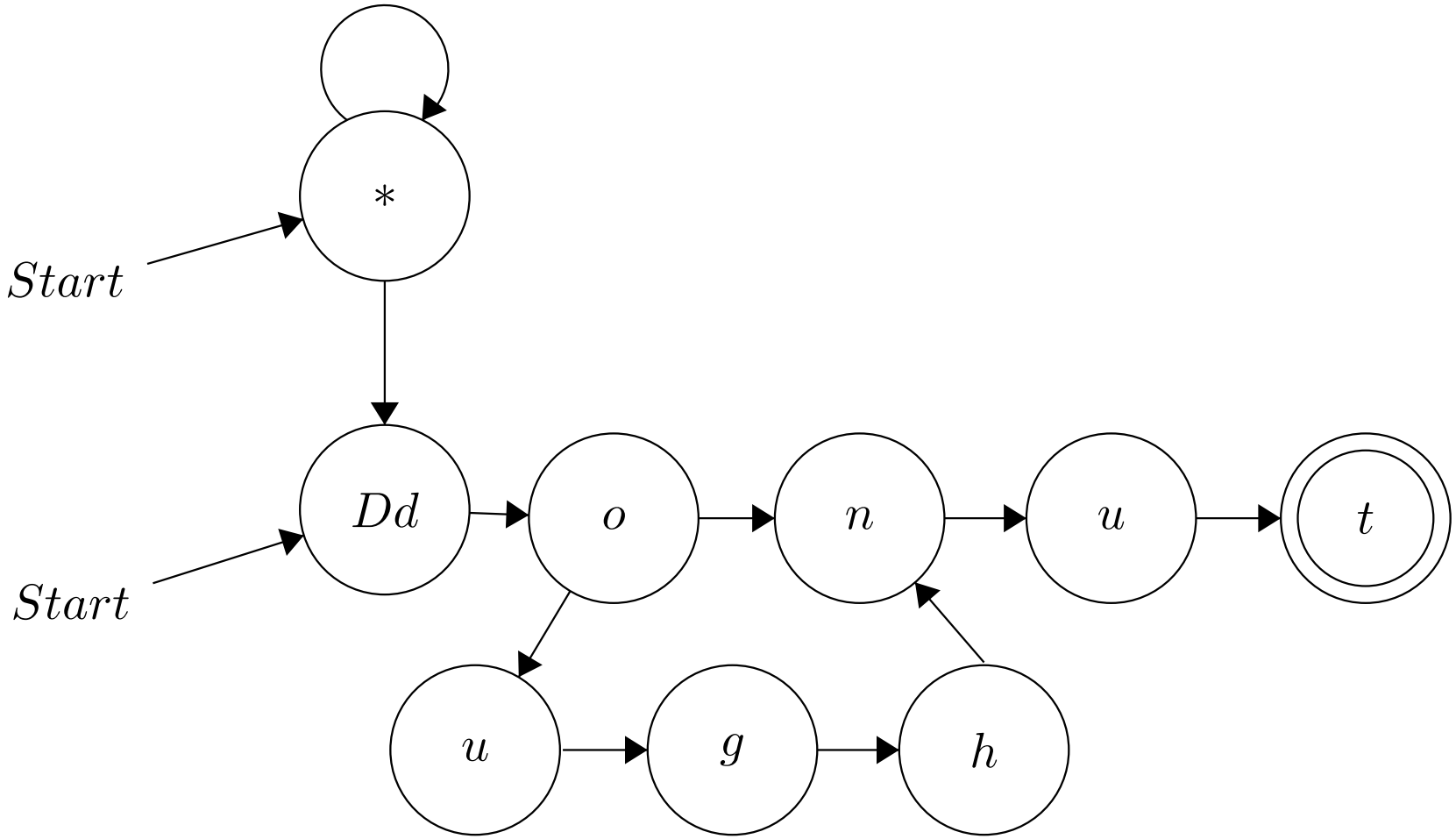
Check: 1110

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?



Let … NFA

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

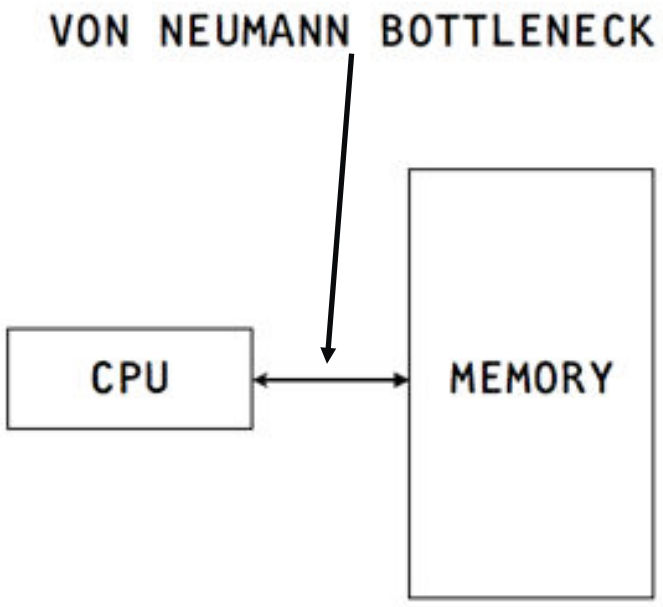
C… …FA

	Processing Complexity	Storage Cost
NFA	$O(n^2)$	$O(n)$
DFA	$O(1)$	$O(\sum^n)$

Processing Bandwidth

Memory Bandwidth

Architectural Underpinnings



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

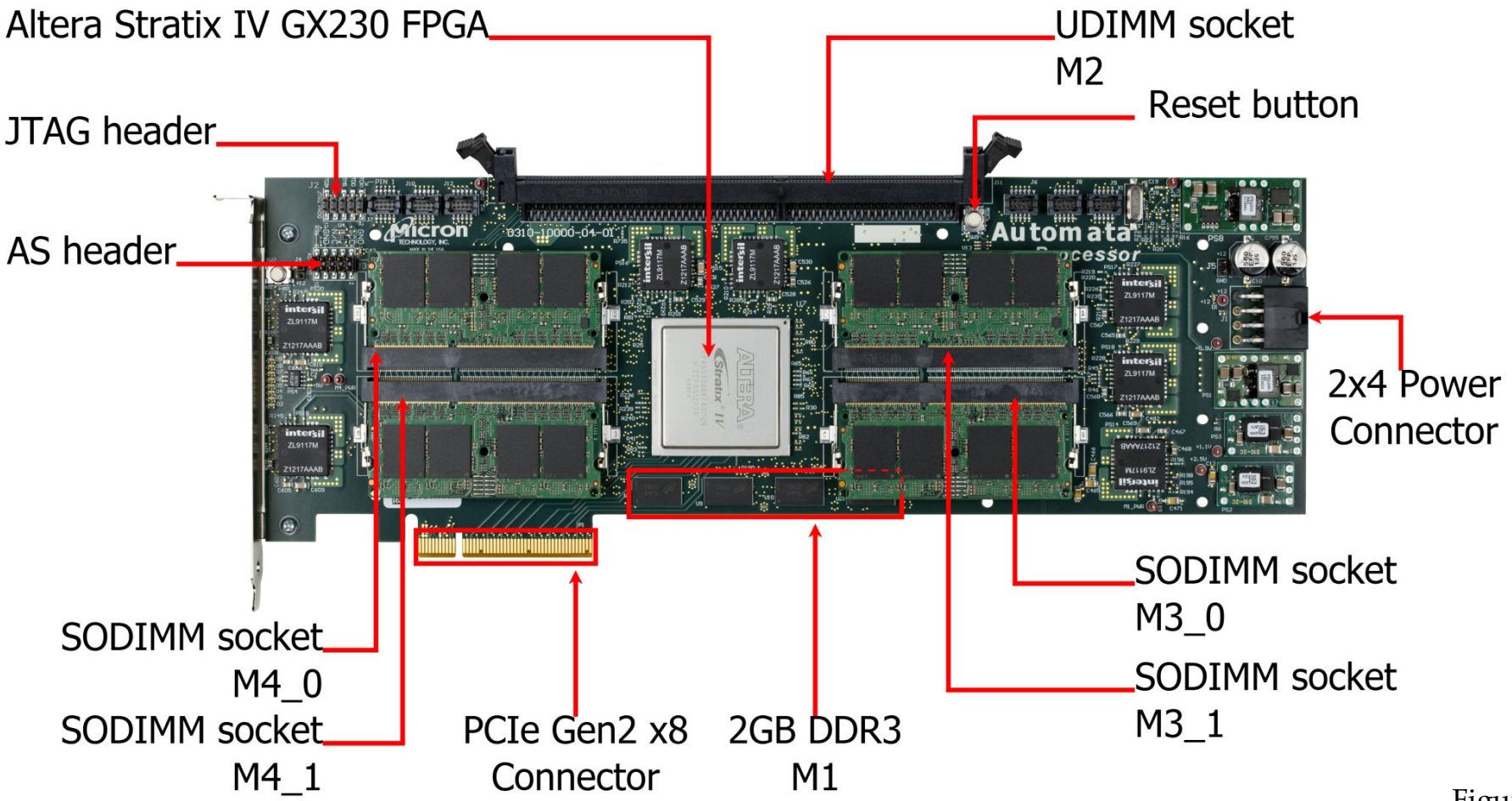


Figure courtesy of Micron

Exploiting DRAM

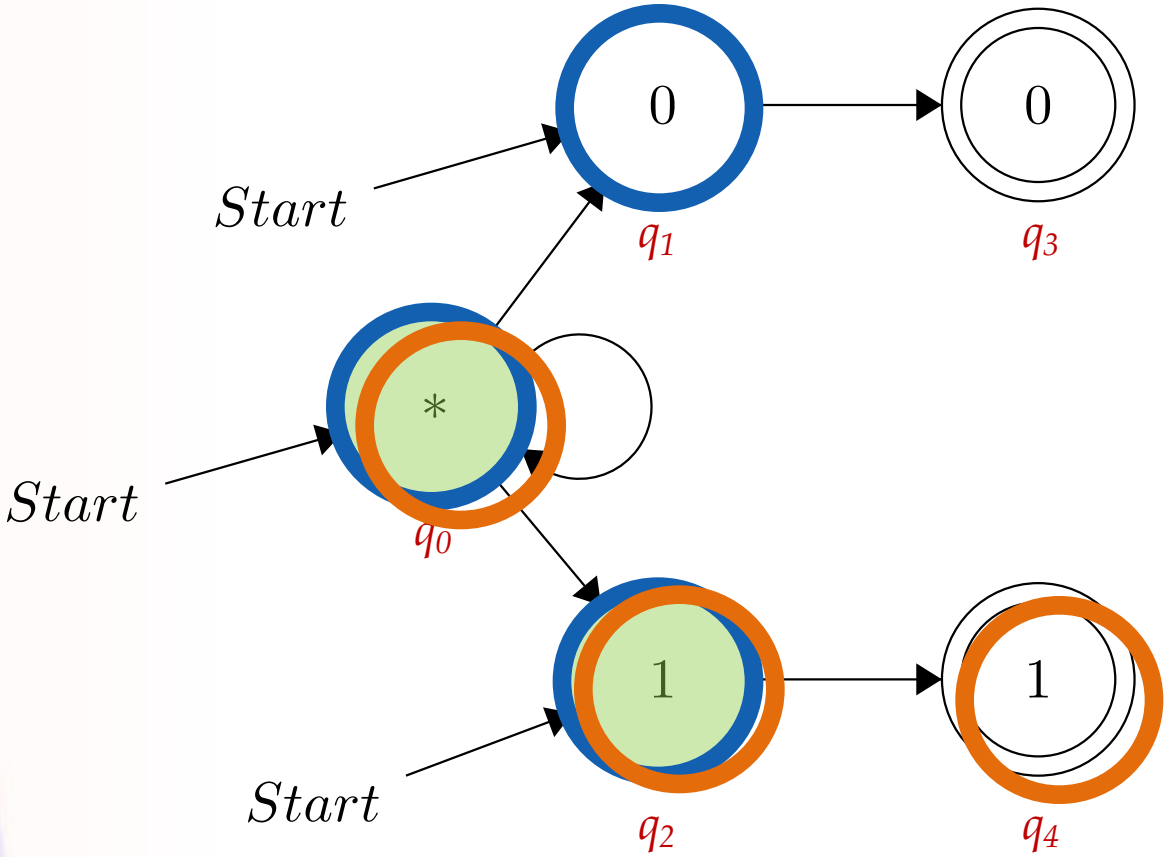
Row Access results in **one** word being retrieved from memory

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

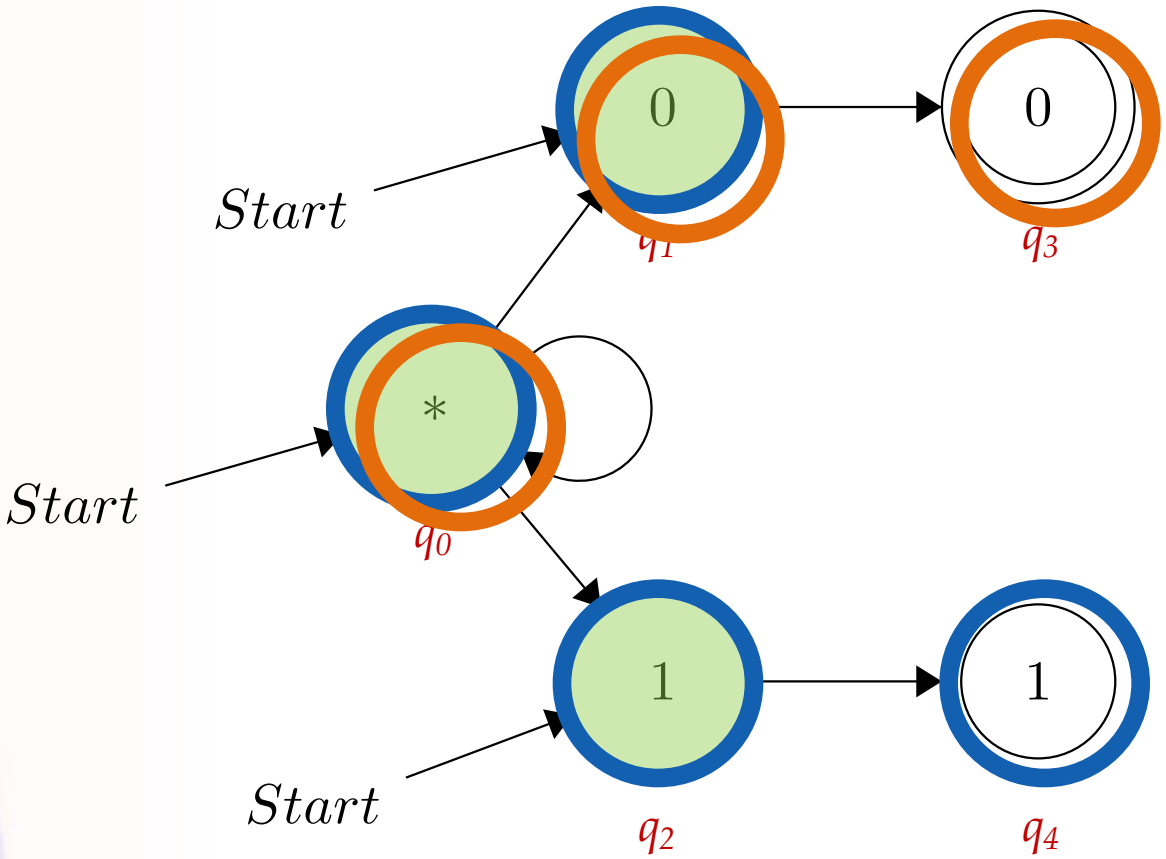
Input Stream: 1011



- 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

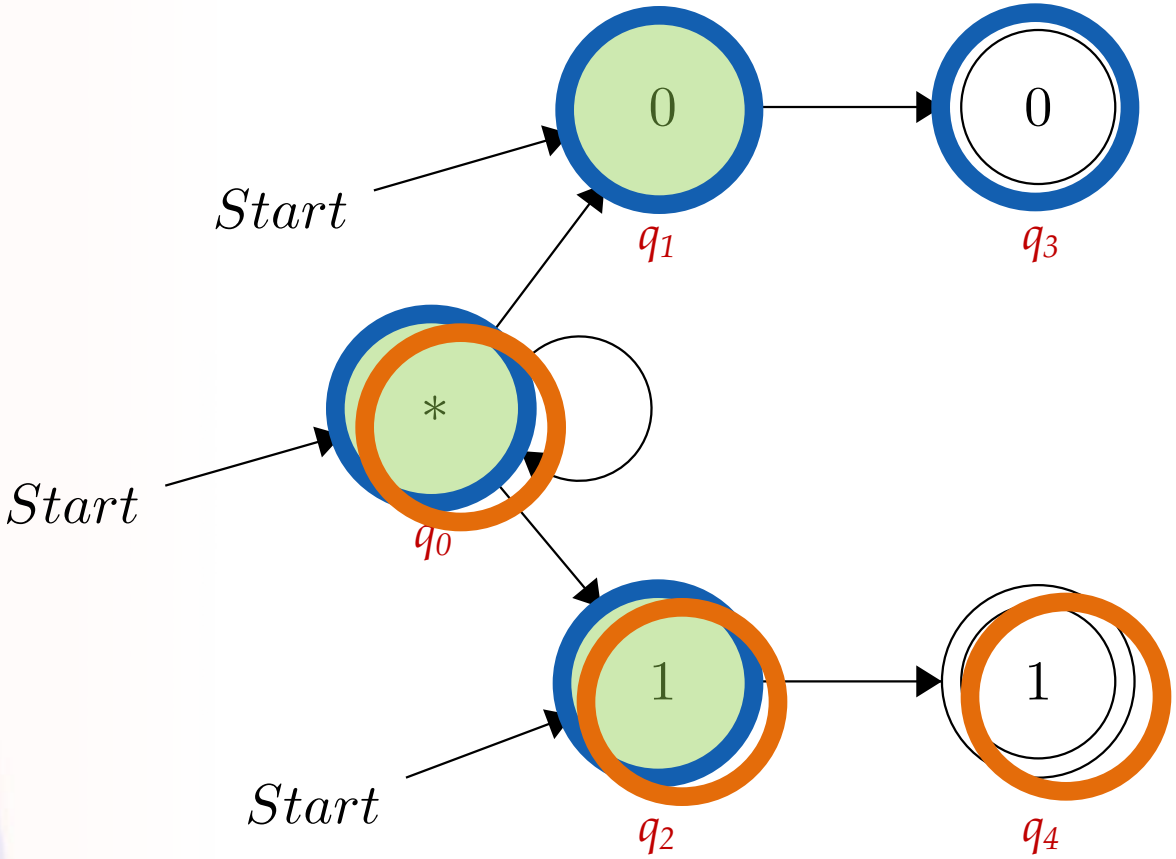
Input Stream: 1011



- 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

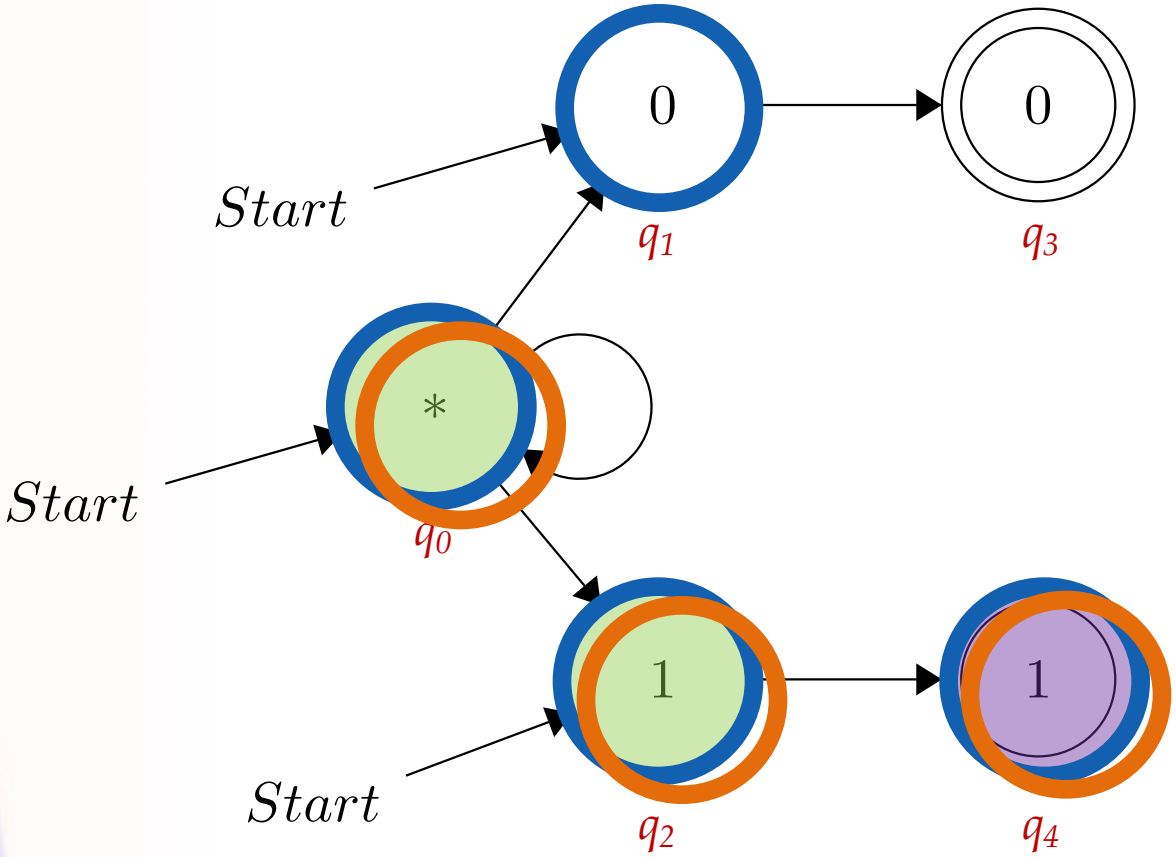
Input Stream: 1011



- Bit-parallel closure({q_0, q_2}):
  - {q_0, q_1, q_2, q_3}
- Symbol closure(1):
  - {q_0, q_2, q_4}
- Bit & Symbol:
  - {q_0, q_2, q_4}

Bit-Parallel Execution

Input Stream: 1011



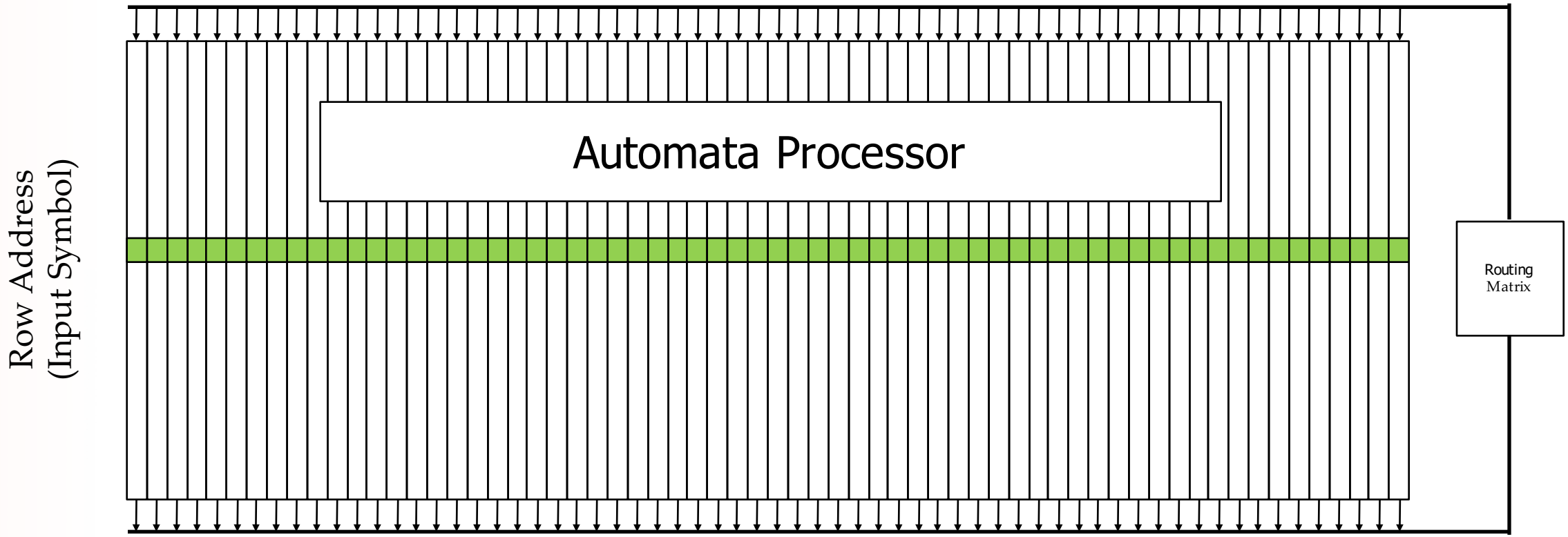
- Bit-parallel closure({q_0, q_2}):
  - {q_0, q_1, q_2, q_4}
- Symbol closure(1):
  - {q_0, q_2, q_4}
- Bit & Symbol:
  - {q_0, q_4}

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

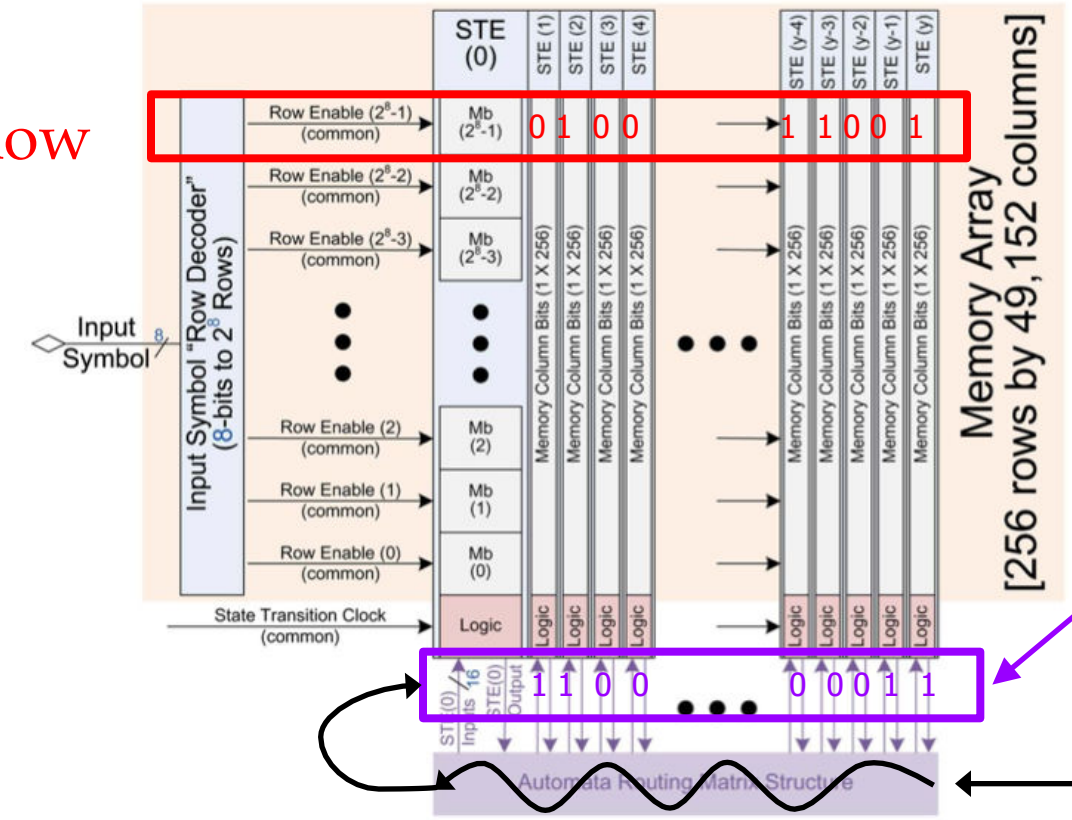


Row Access results in **49,152** match & route operations

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



Columns with "1": STEs that accept input symbol

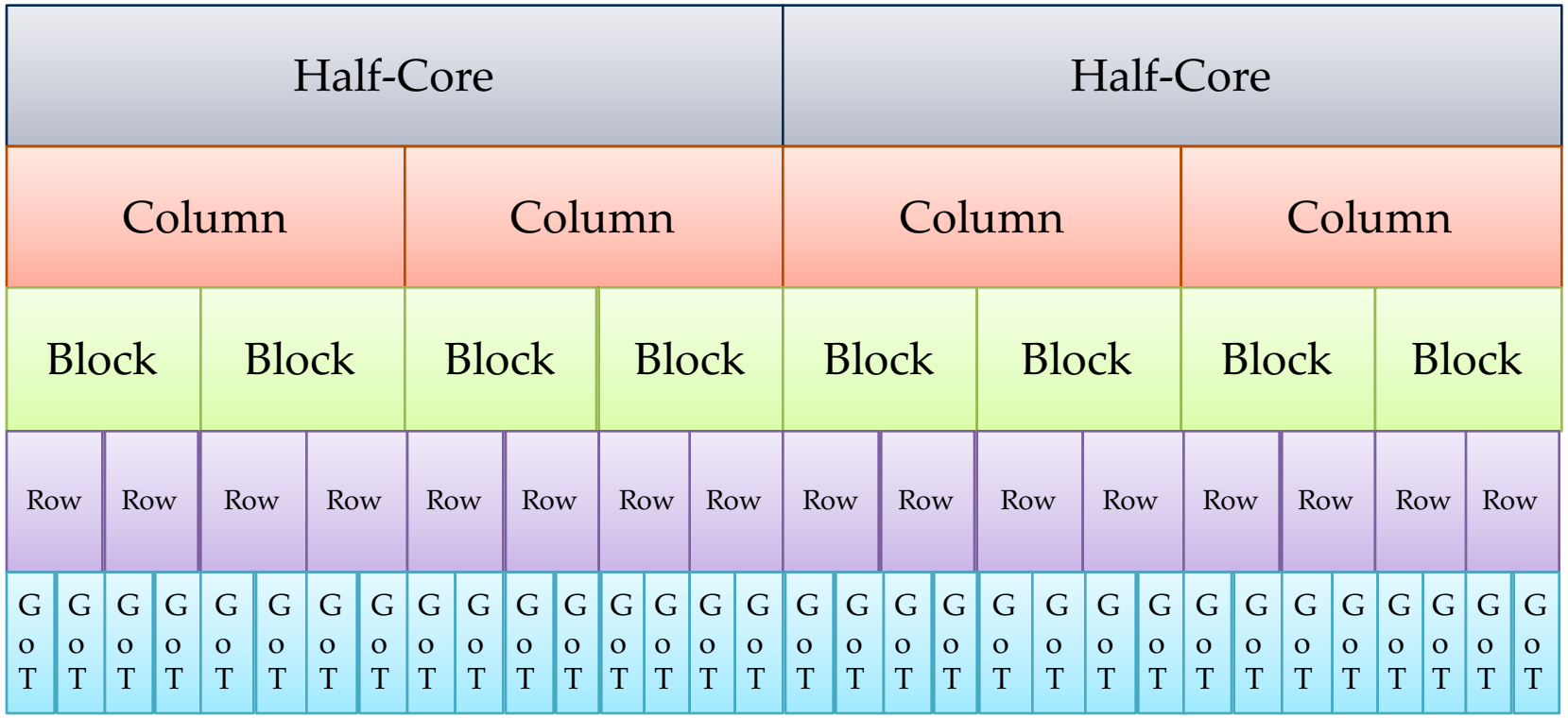
&

Active States

=

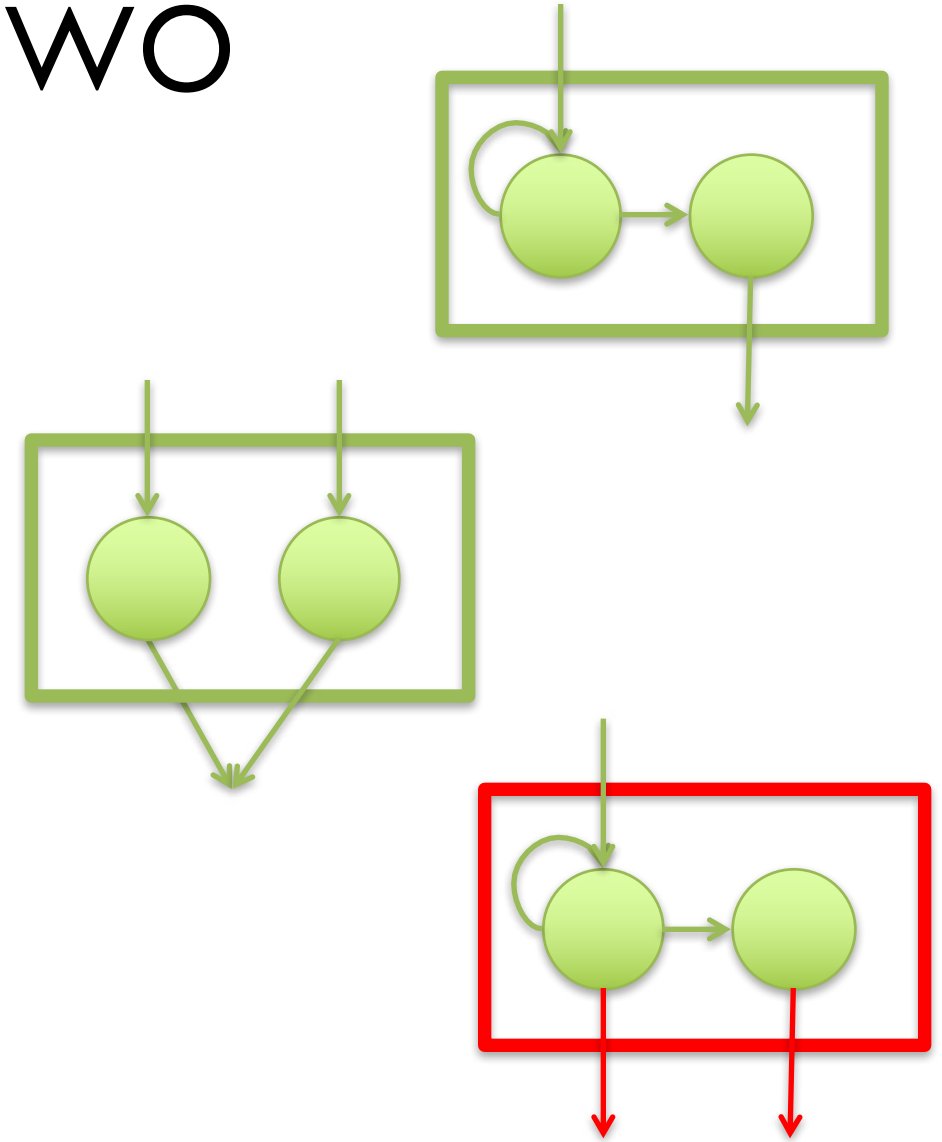
Active States for Next Clock Cycle

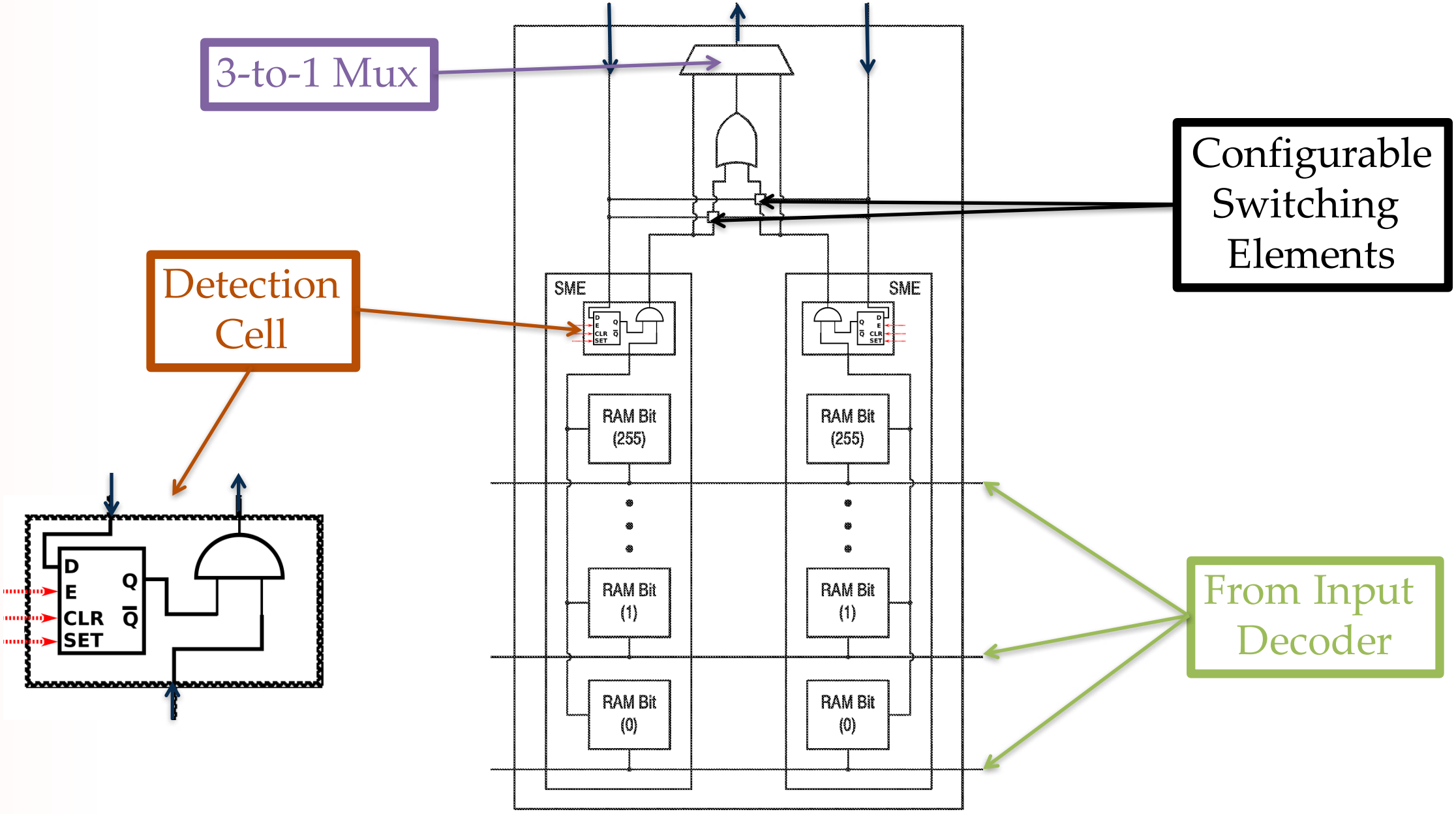
Why Hierarchical Routing?



Group of Two

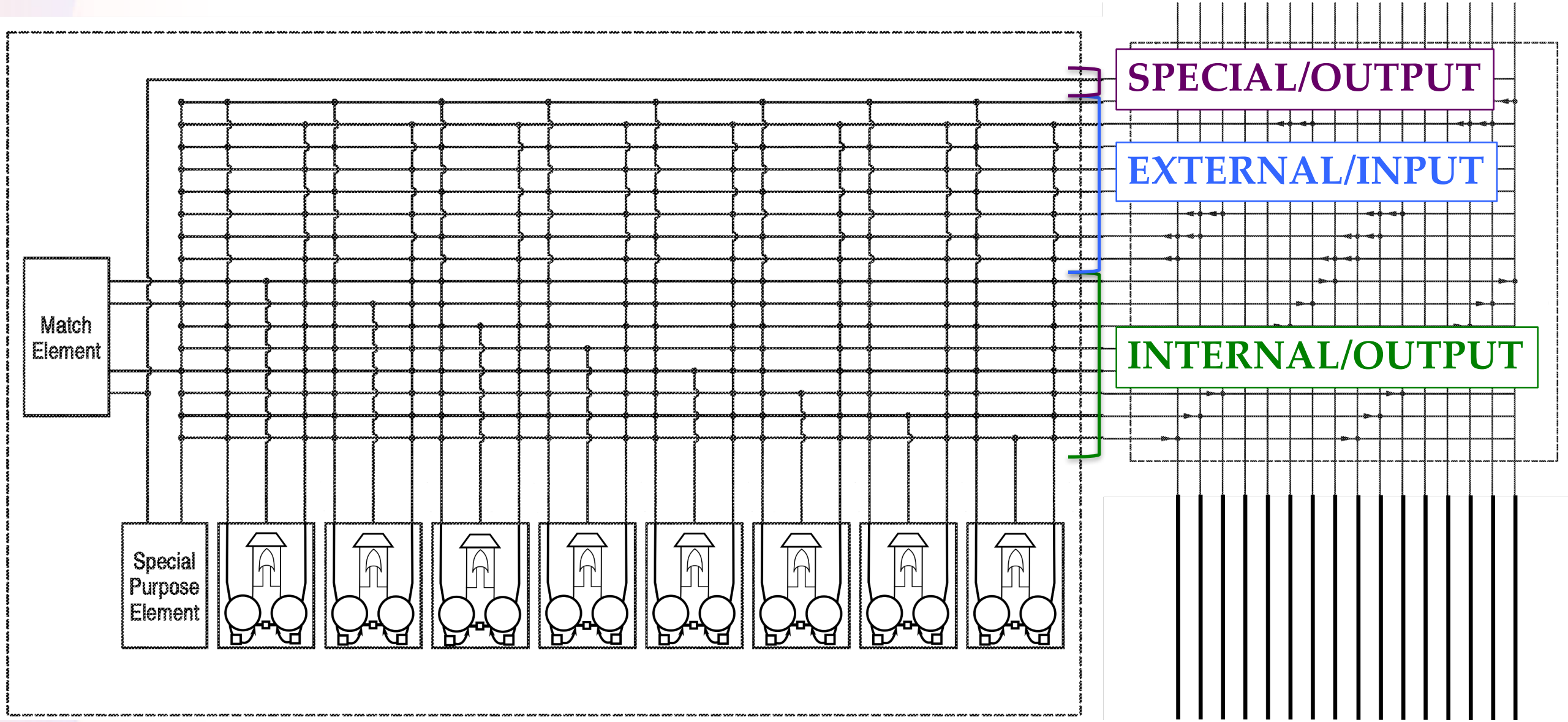
- Contain two State Machine Elements (SME)
  - Hardware embodiment of STE
  - Memory Cells
  - Detection Cell
- Common output
- Loopback and Chaining




3-to-1 Mux
Configurable Switching Elements
Detection Cell
SME
SME
RAM Bit (255)
RAM Bit (255)
RAM Bit (1)
RAM Bit (1)
RAM Bit (0)
RAM Bit (0)
From Input Decoder
D
E
CLR
SET
Q
Q̄

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)


SPECIAL/OUTPUT
EXTERNAL/INPUT
INTERNAL/OUTPUT
Match Element
Special Purpose Element

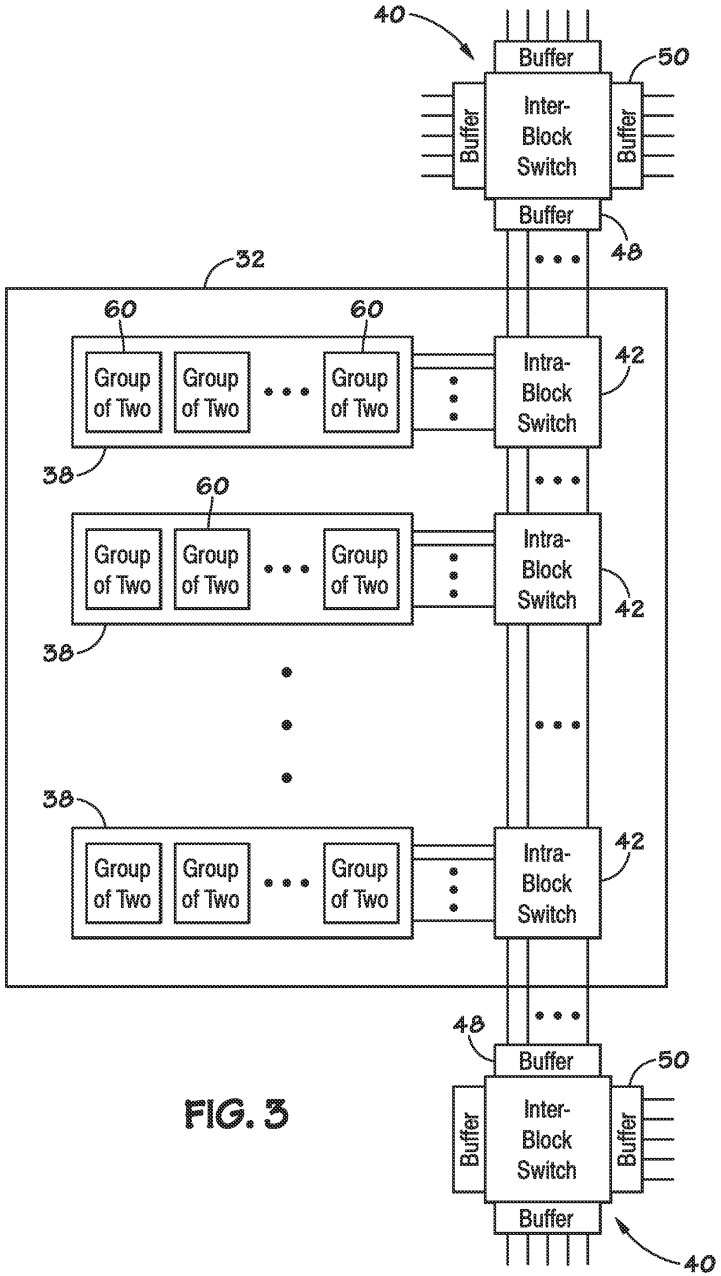


FIG. 3

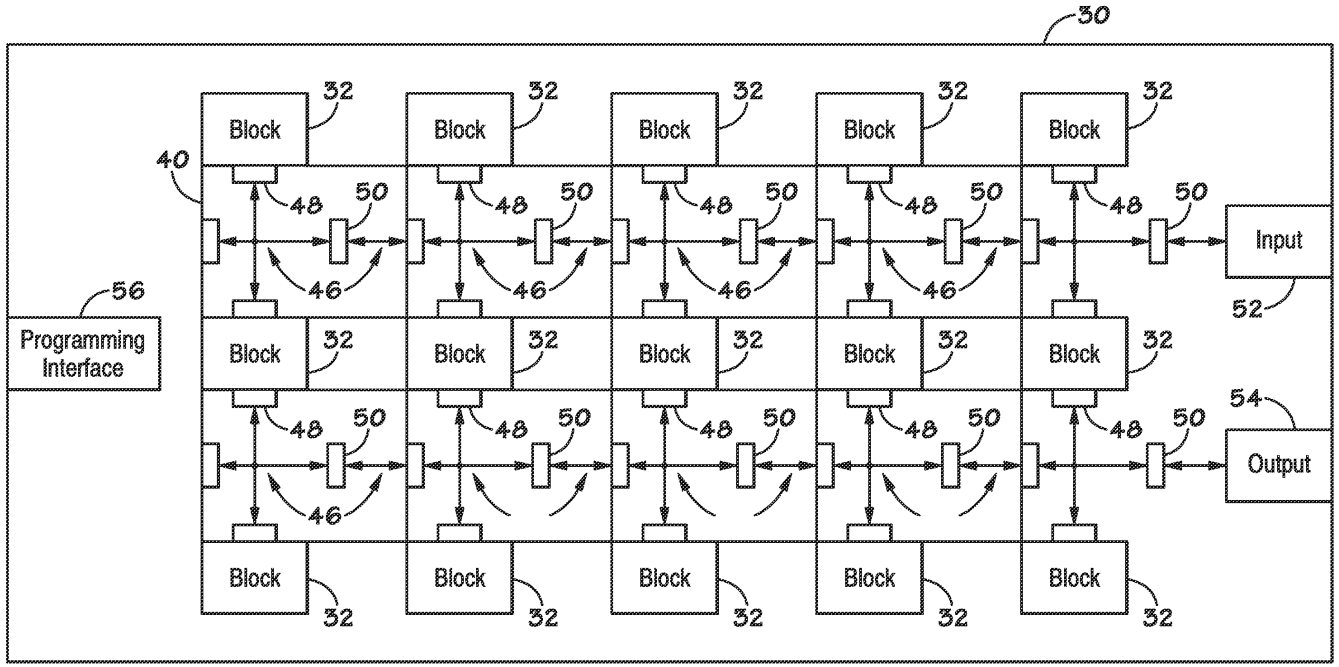
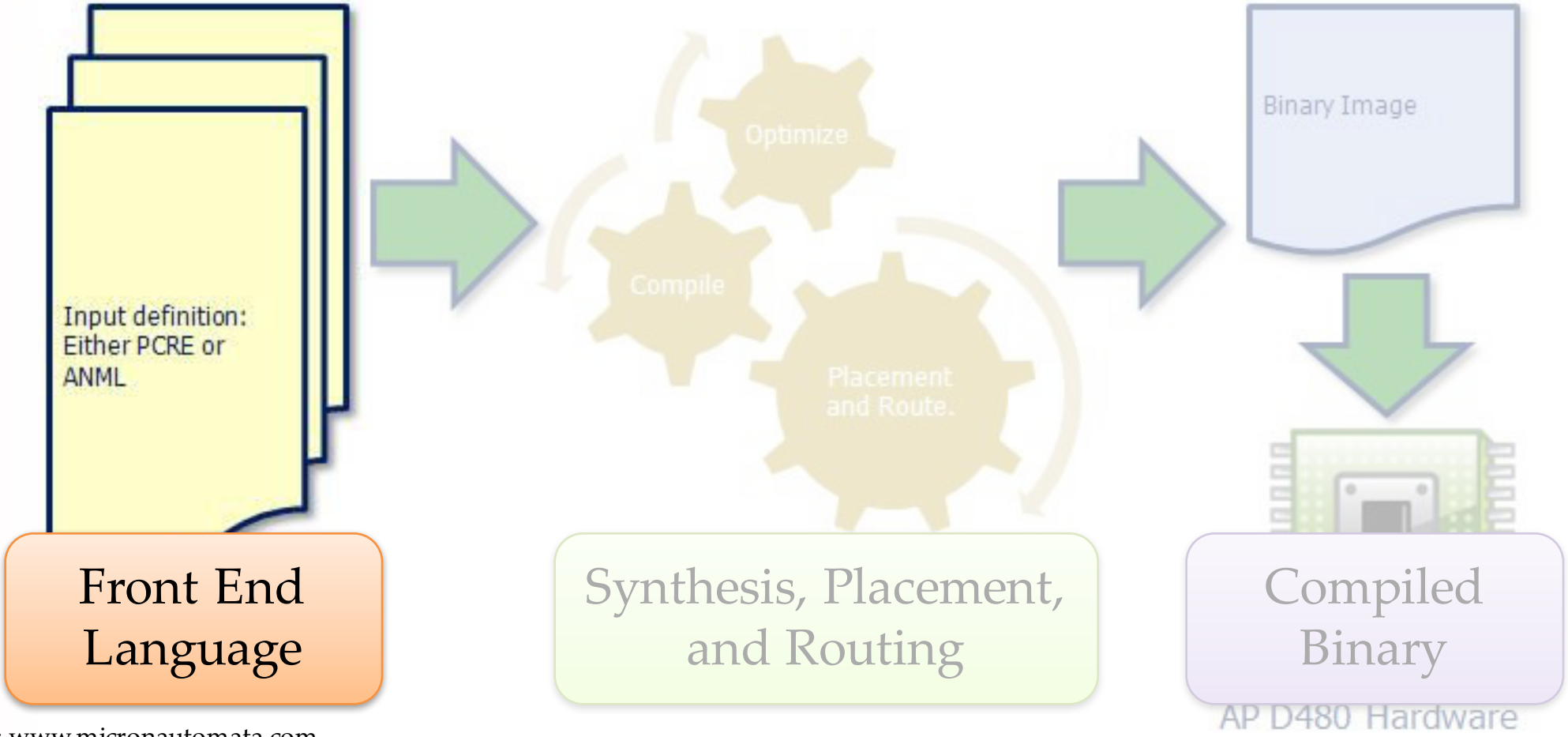


FIG. 2

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

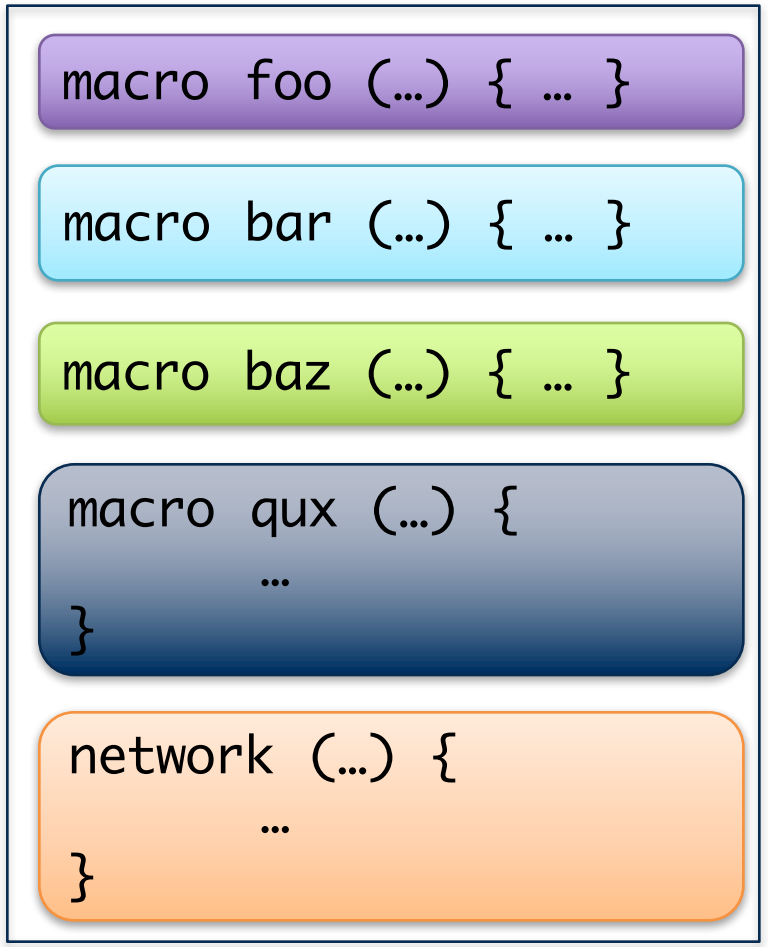
- Pattern-Based Data Analysis
- Automata Processor
- Current Programming Models
- RAPID Language Overview
- AP Code Generation and Optimizations
- Evaluation

RAPID at a Glance

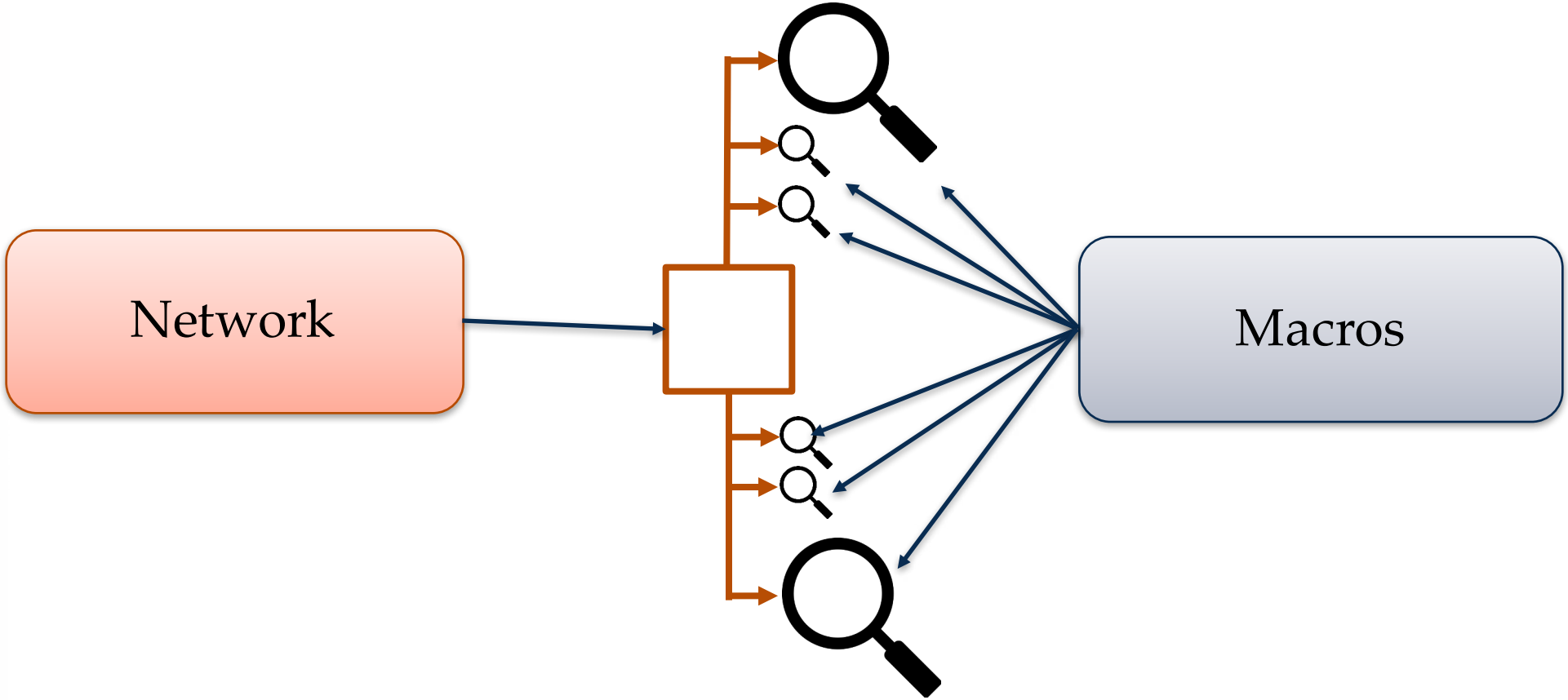
- Provides concise, maintainable, and efficient representations for pattern-identification algorithms
- Conventional, C-style language with domain-specific parallel control structures
- Excels in applications where patterns are best represented as a combination of text and computation
- Compilation strategy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

Program Structure

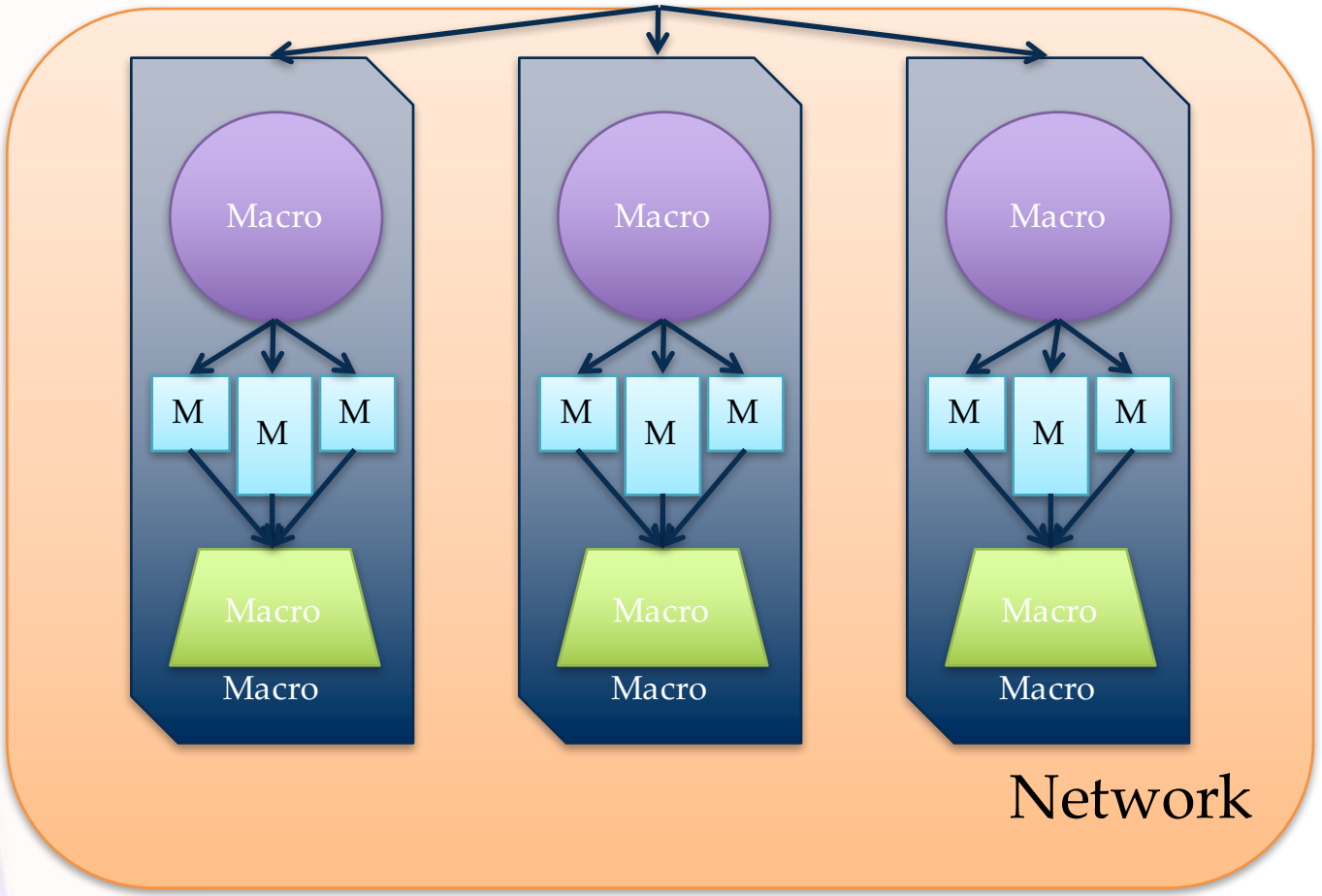
- **Macro**
  - Basic unit of computation
  - Sequential control flow
  - Boolean expressions as statements for terminating threads of computation
- **Network**
  - High-level pattern matching
  - Parallel control flow
  - Parameters to set run-time values



Program Structure



Program Structure




```
macro foo (…) { … }

macro bar (…) { … }

macro baz (…) { … }

macro qux (…) {
        …
}

network ( … ) {
        …
}
```

Data in RAPID

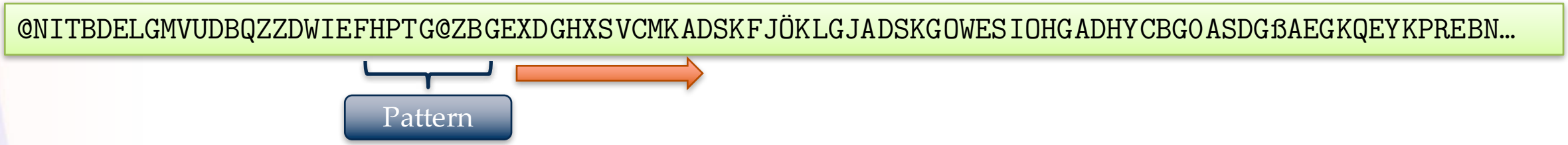
- Input data stream as special function
  - Stream of characters
  - `input()`
    - Calls to `input()` are synchronized across all active macros
    - All active macros receive the same input character

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

Sequential Structure	Parallel Structure
if…else	either…orelse
foreach	some
while	whenever

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

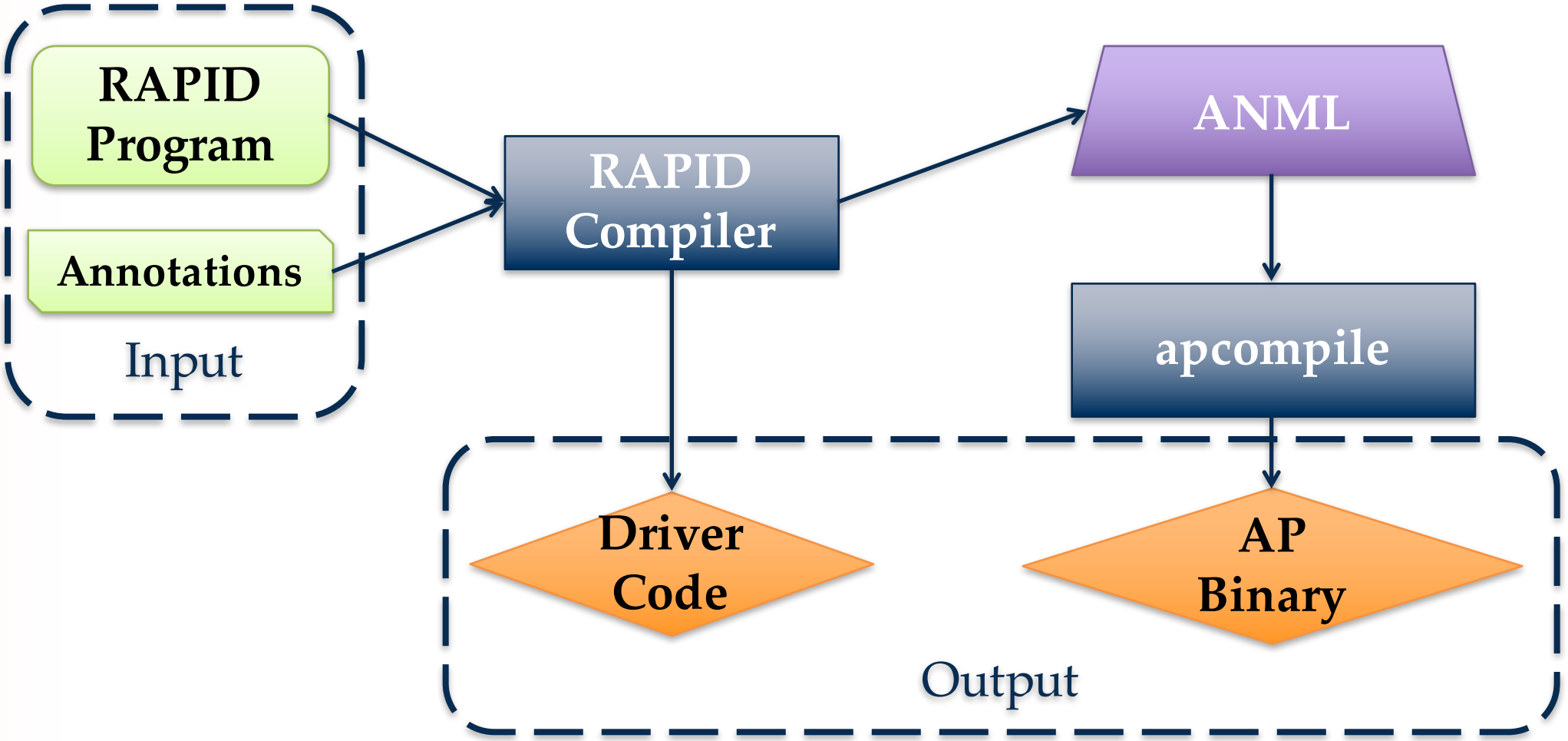
```
macro frequent (String set, Counter cnt) {
        foreach(char c : set) {
            while(input() != c);
        }
        cnt.count();
}

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

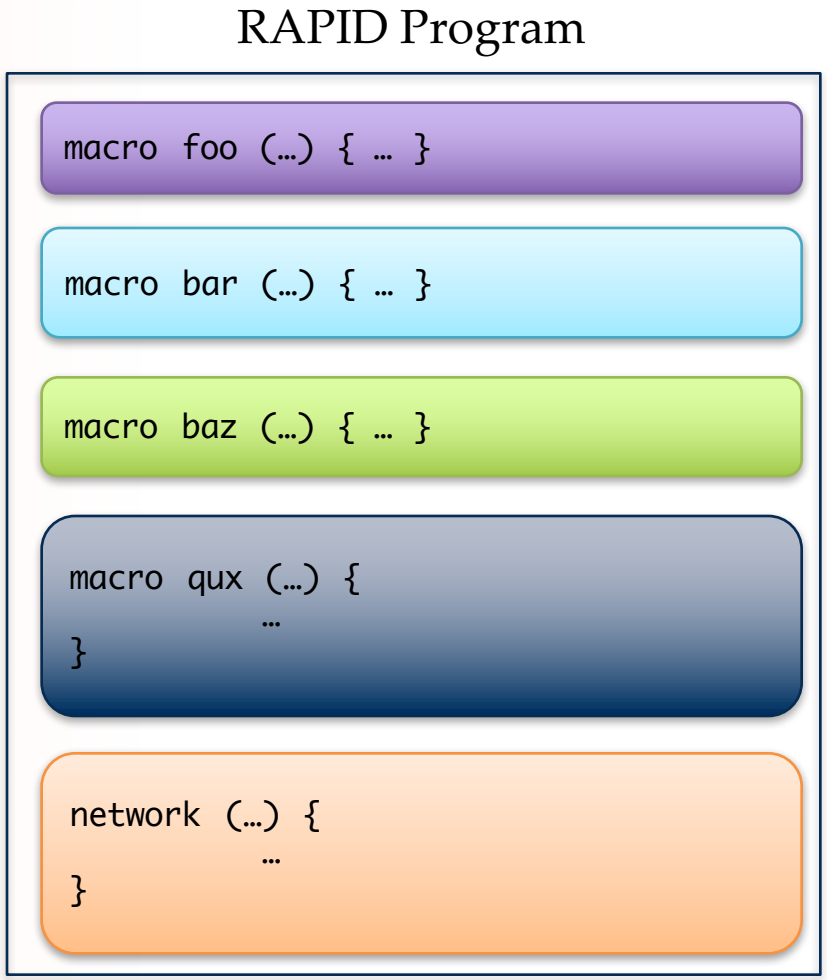
RAPID Programming

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

System Overview

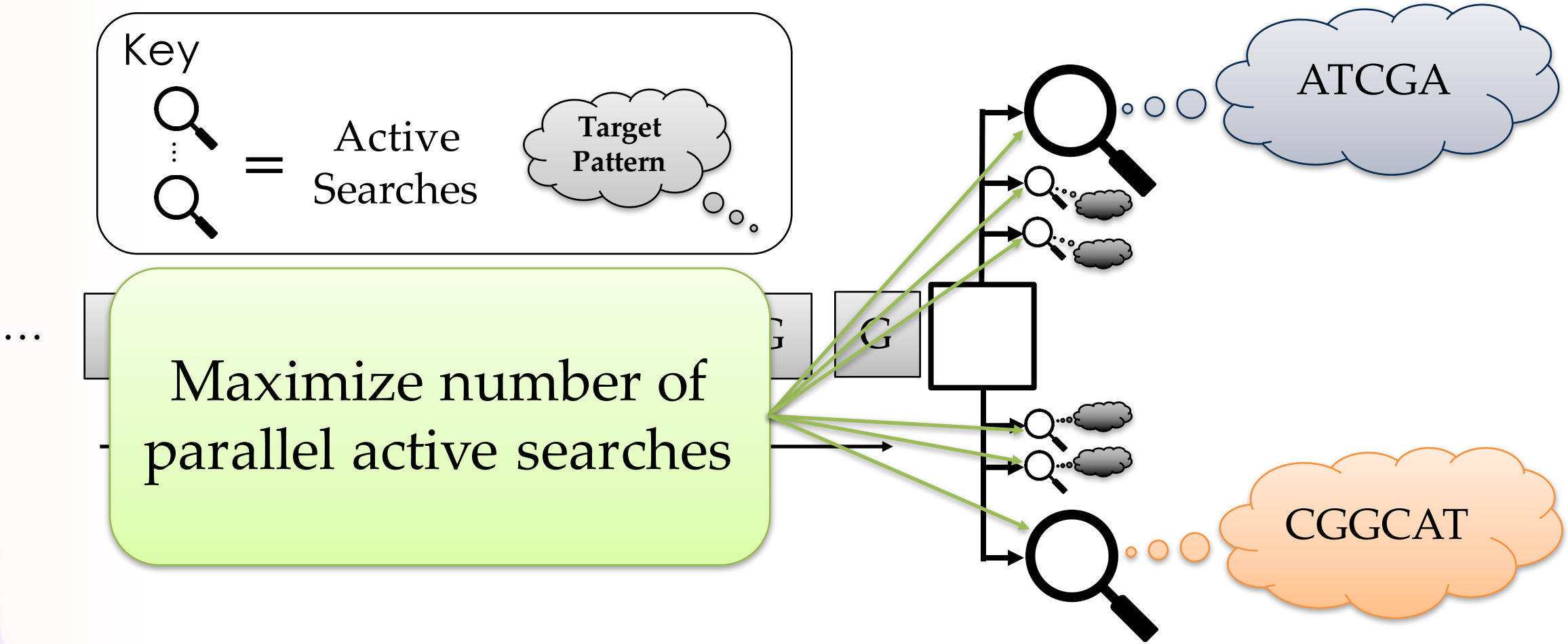


Code Generation



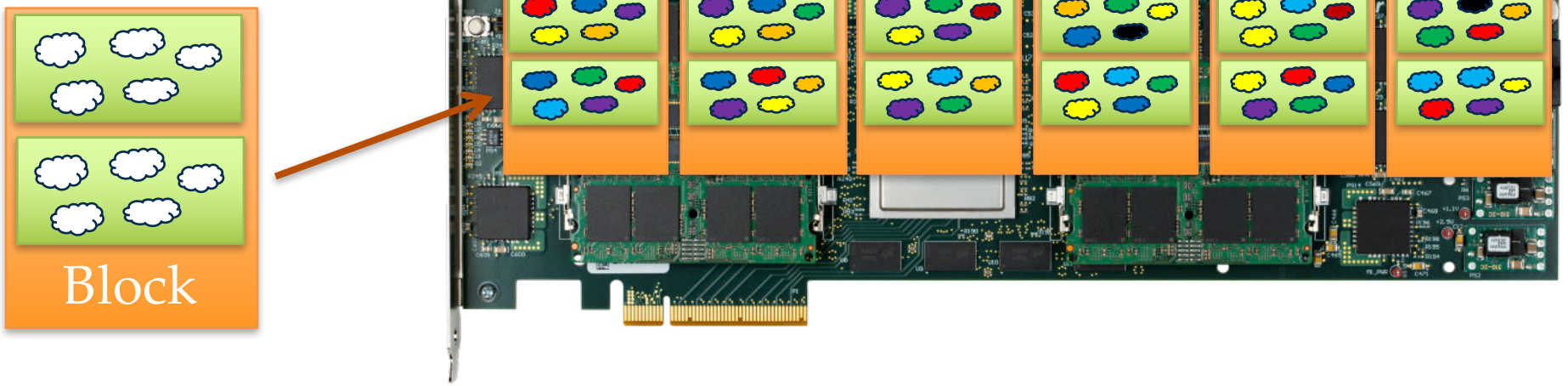
- Recursive transformation of RAPID program
  - Input Stream → STEs
  - Counters → 1 or more physical counter(s)
- Similar to RegEx → NFA transformation

Parallel searches

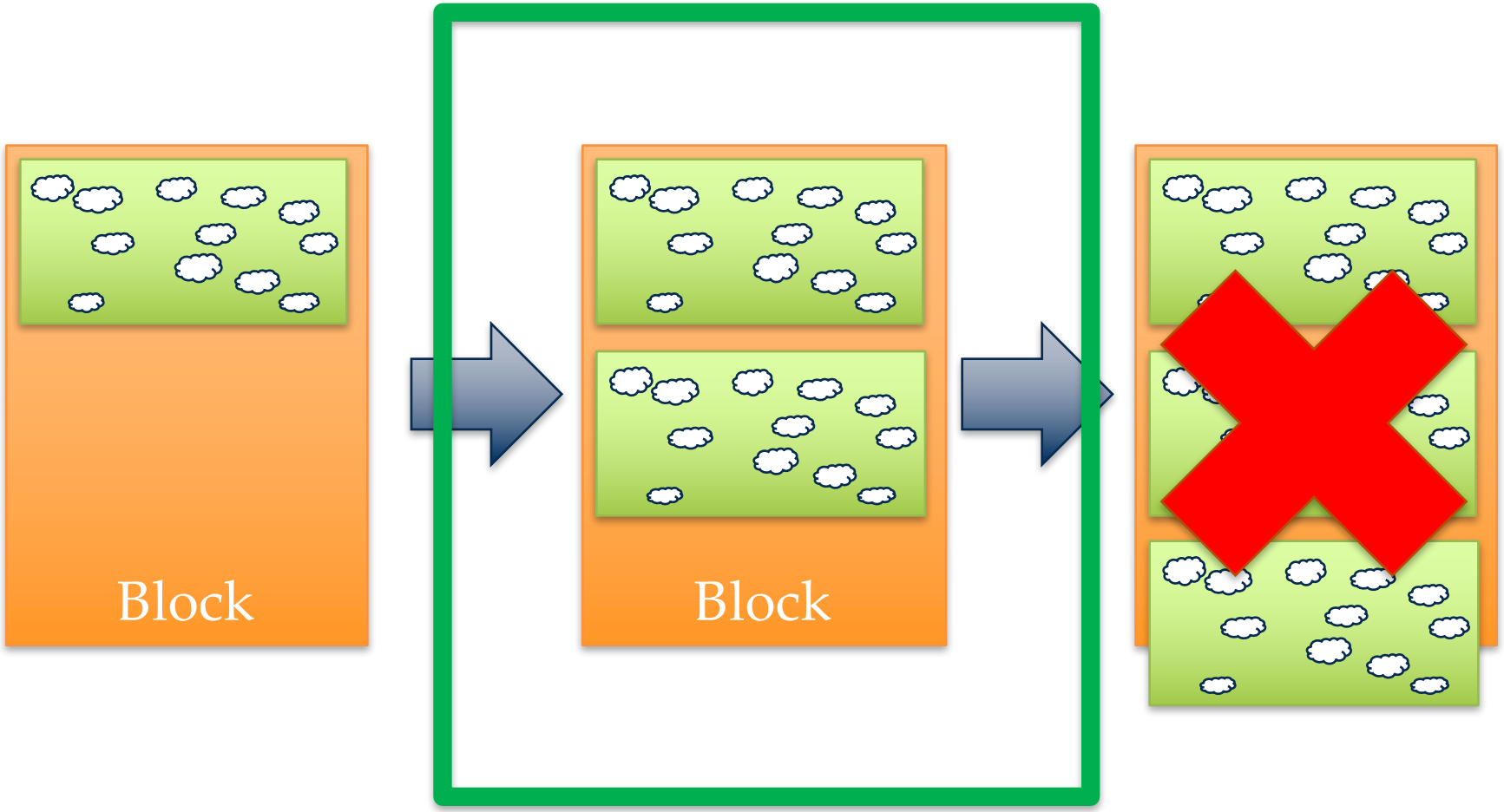


Optimizing Compilation

- 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

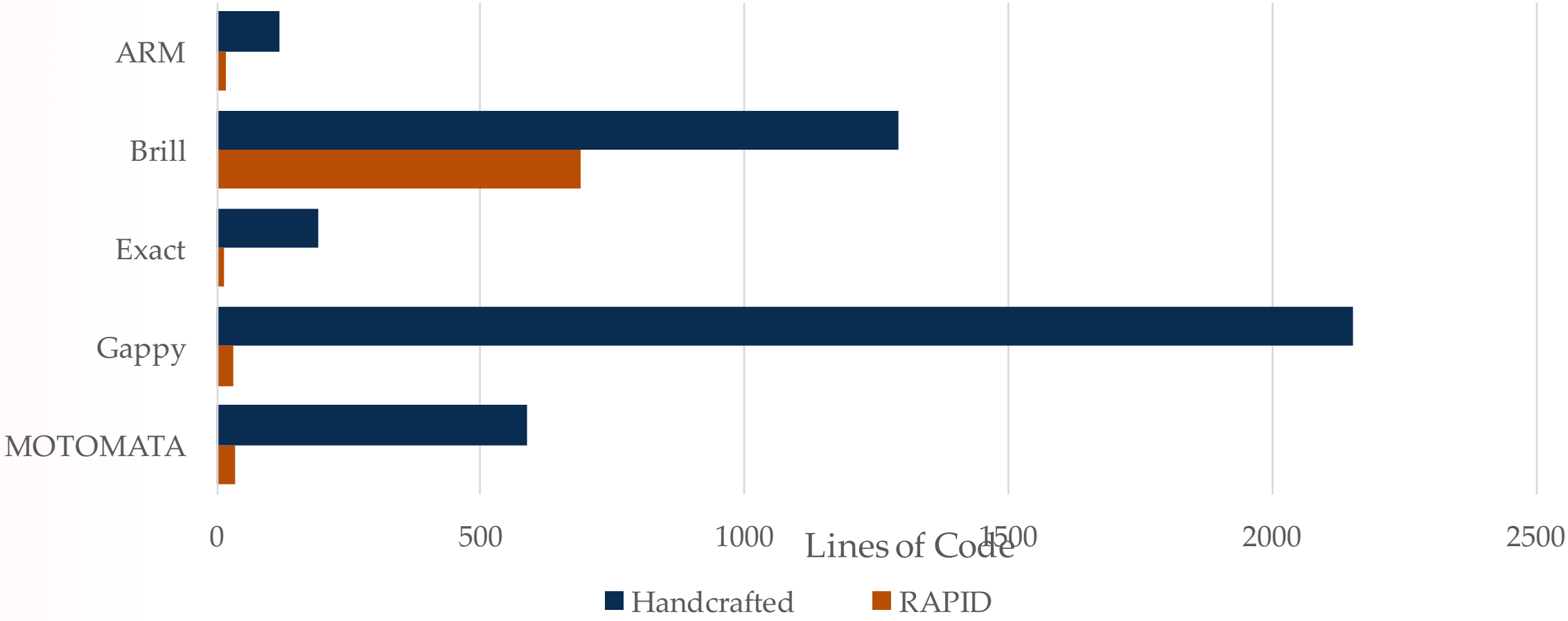
Reminder: Goals

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

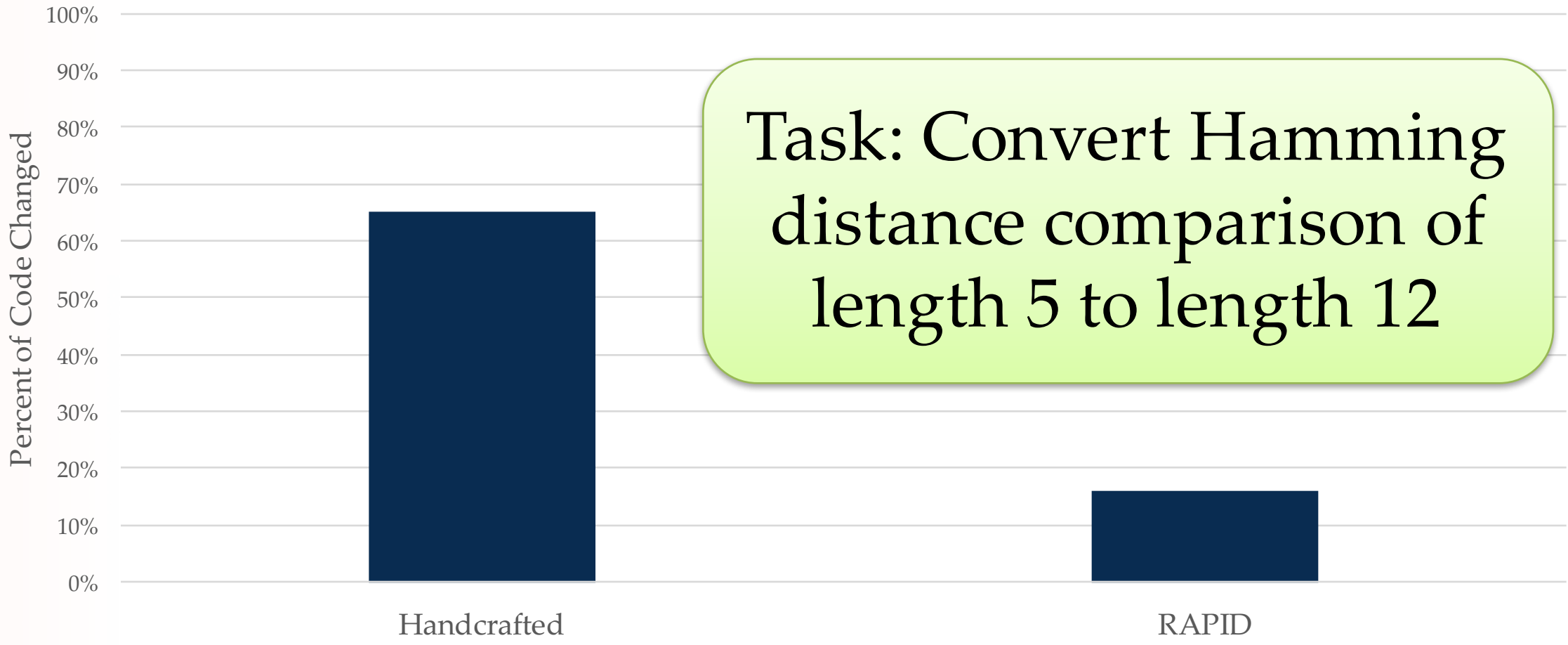
Description of Benchmarks

Benchmark	Description	Generation Method
ARM	Association Rule Mining	Meta Program
Brill	Brill Part of Speech Tagging	Meta Program
Exact	Exact DNA Alignment	ANML
Gappy	DNA Alignment with Gaps	ANML
MOTOMATA	Planted Motif Search	ANML

RAPID Lines of Code

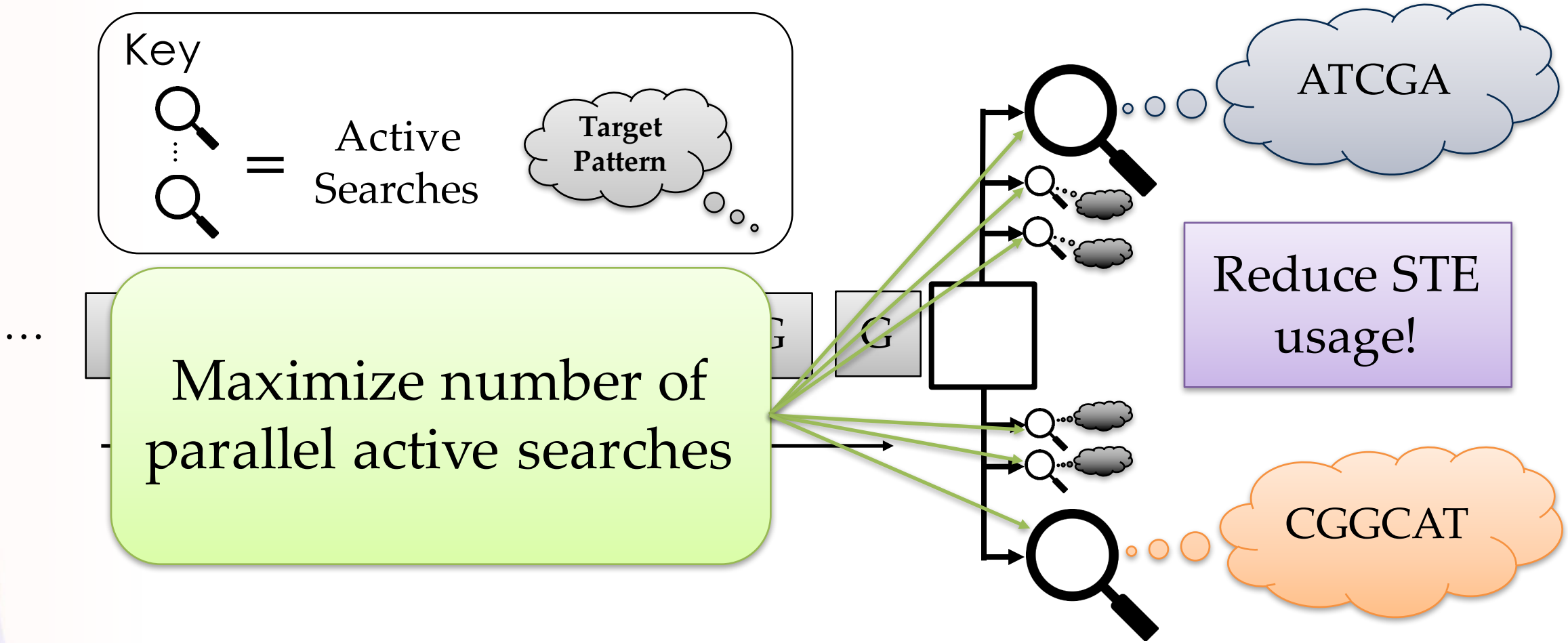


RAPID is Maintainable

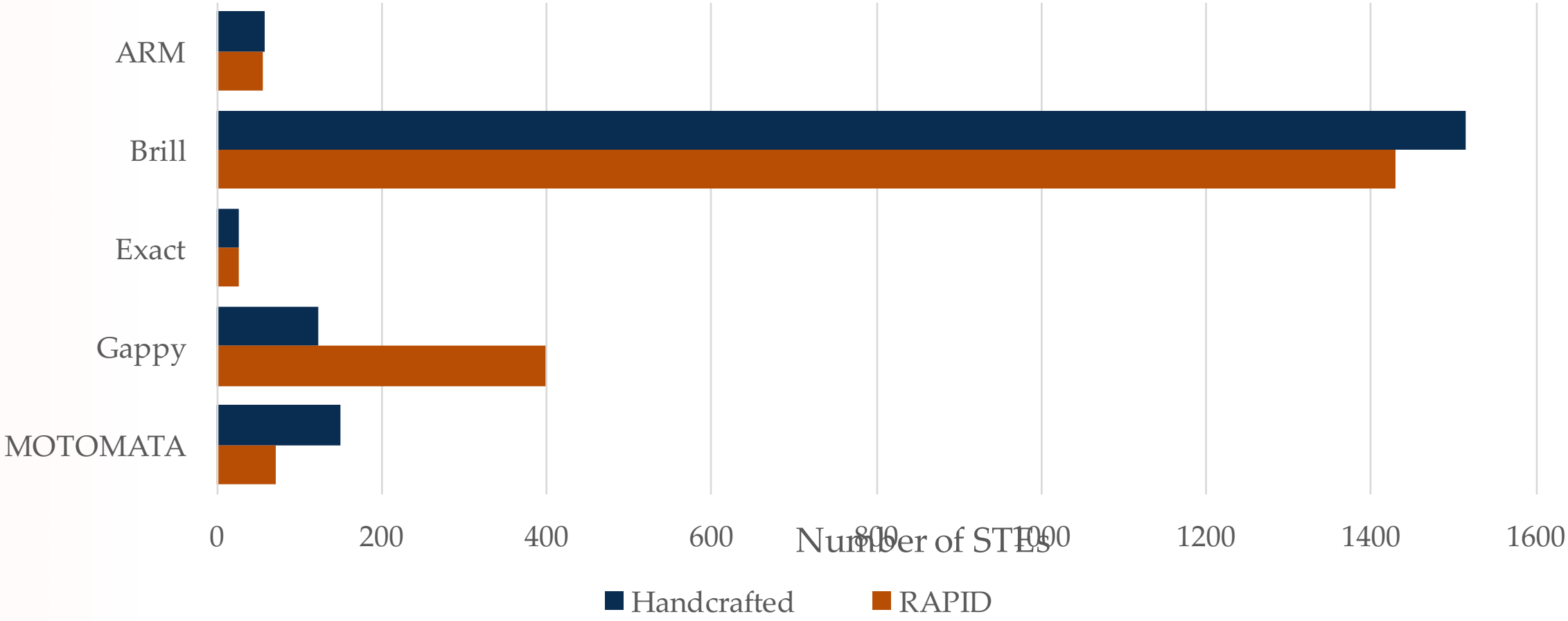


Task: Convert Hamming distance comparison of length 5 to length 12

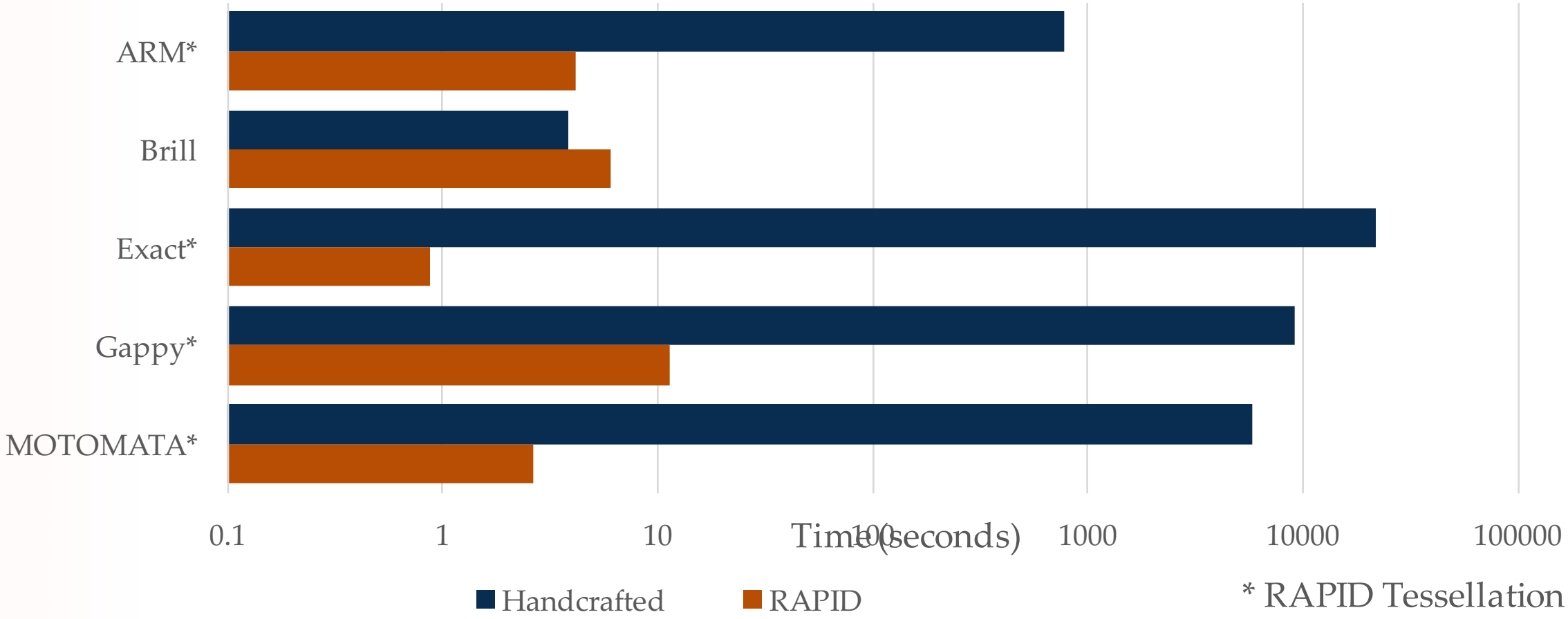
Parallel searches



Generated STEs



Compilation Time



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**