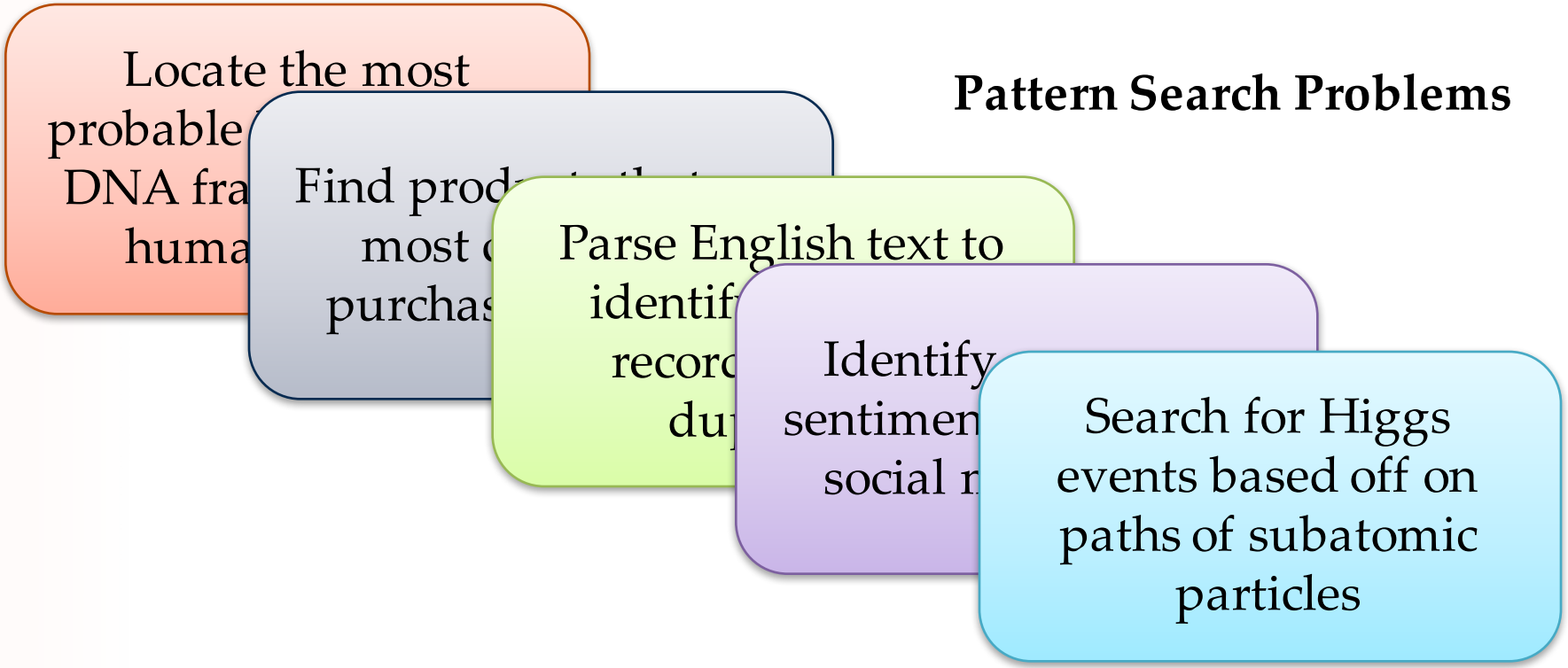
RAPID Programming of Pattern-Recognition Processors

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

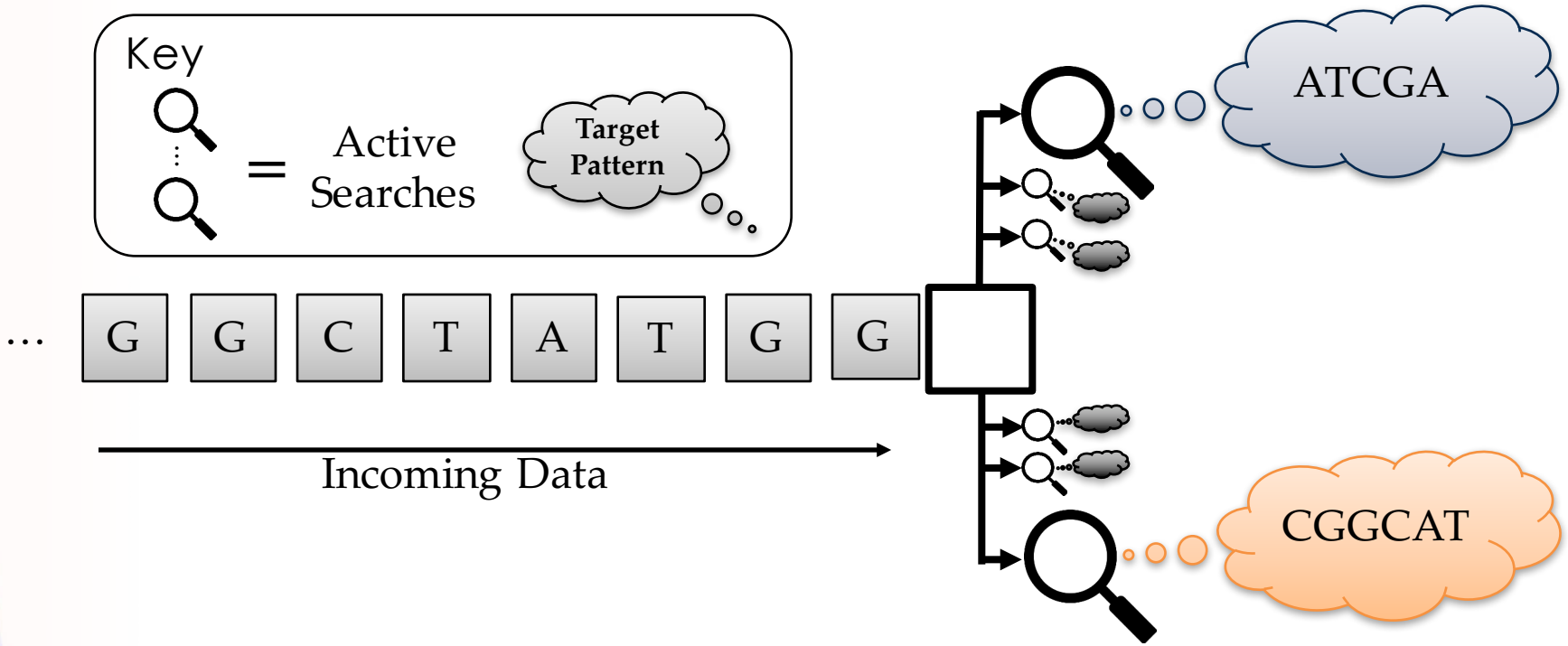
- Researchers and companies are collecting increasing amounts of data
- 44x data production in 2020 than in 2009
- Demand for real-time analysis of collected data

What is the common theme?

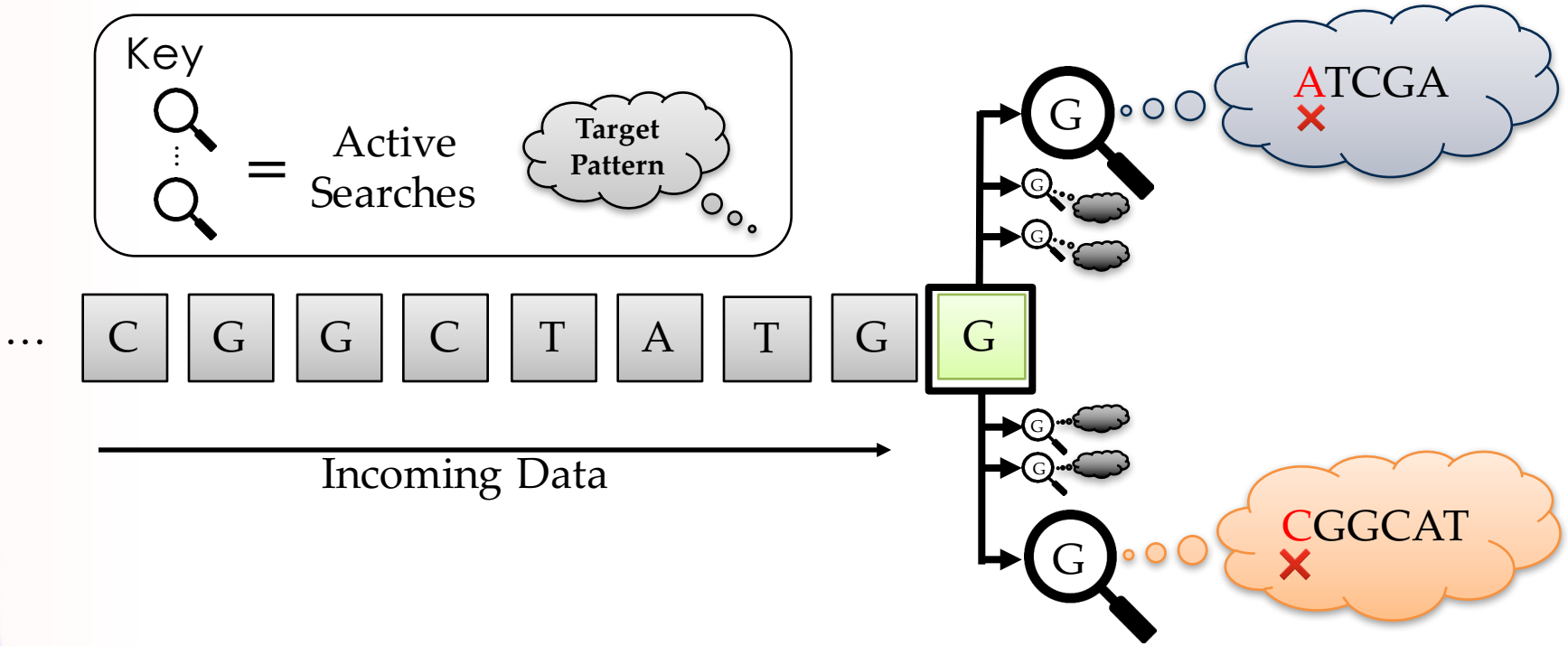


Pattern Search Problems

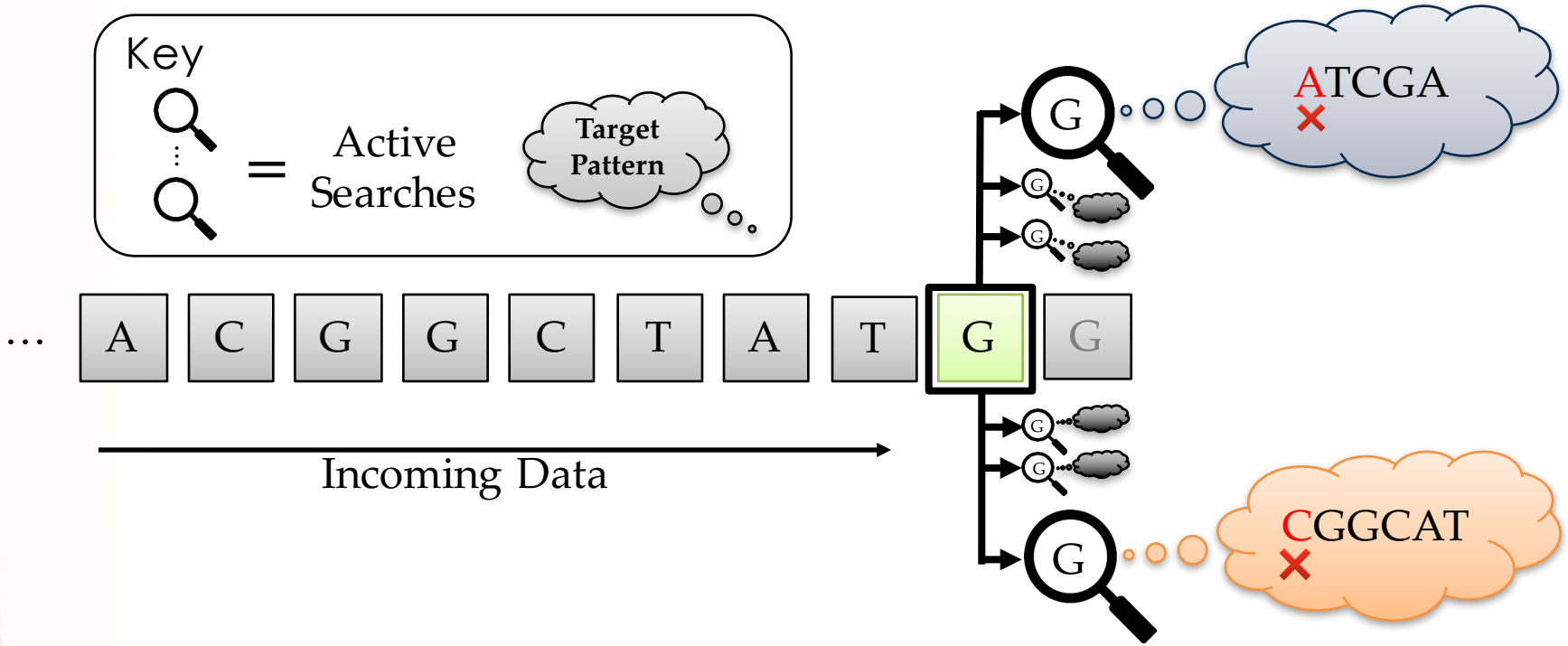
Parallel searches



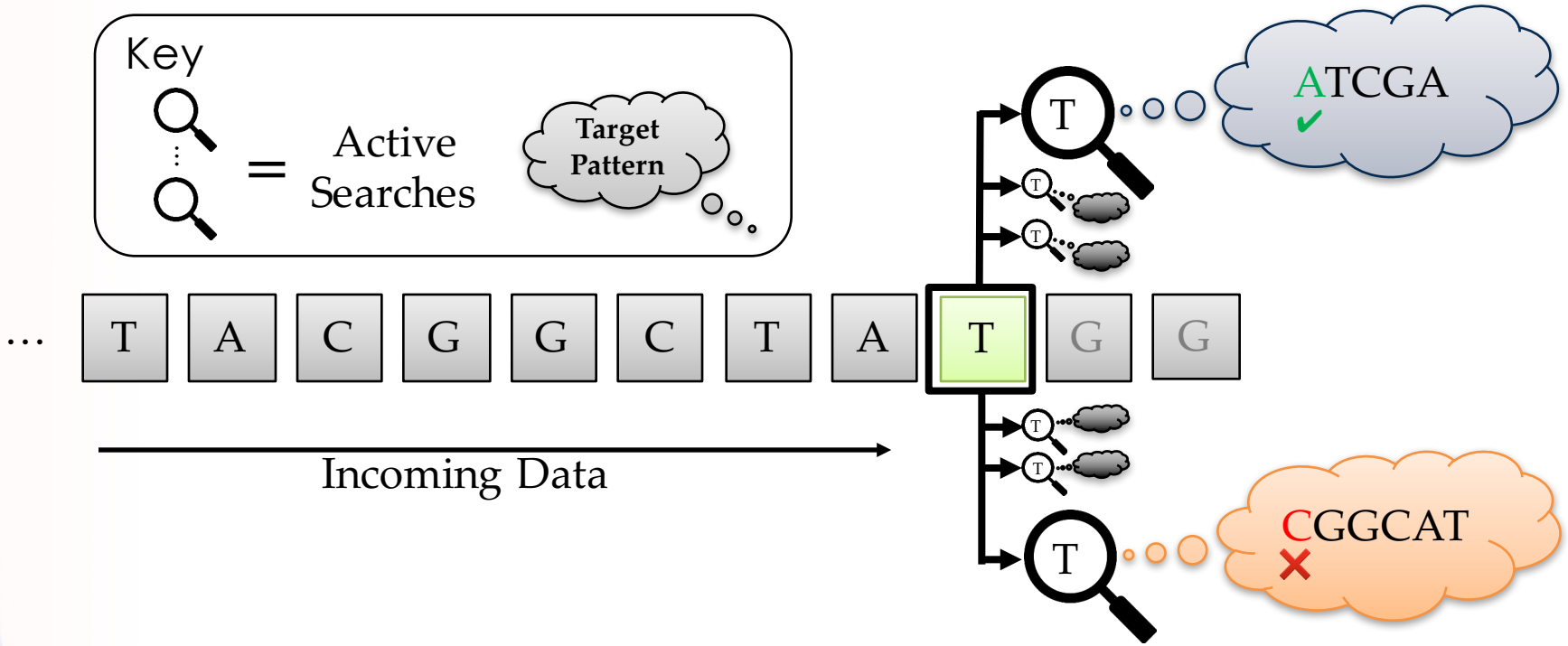
Parallel searches



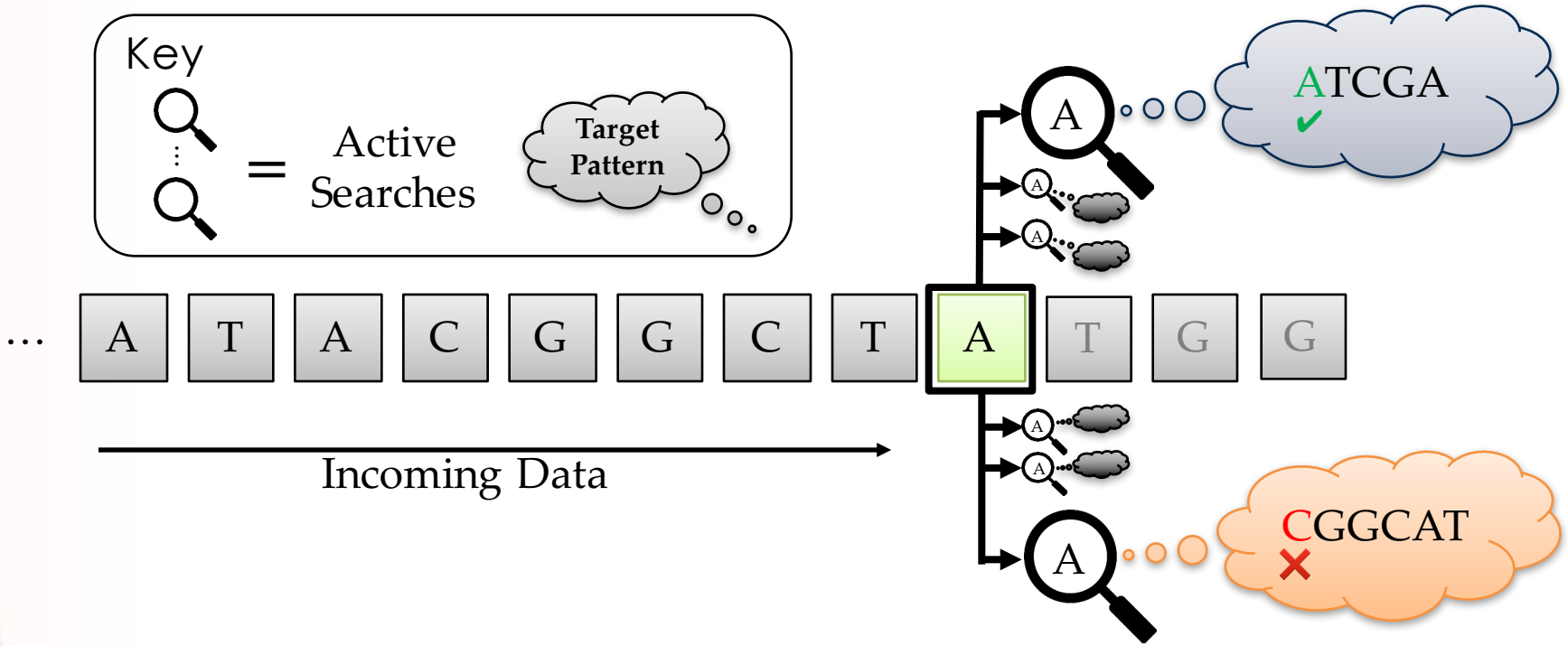
Parallel searches



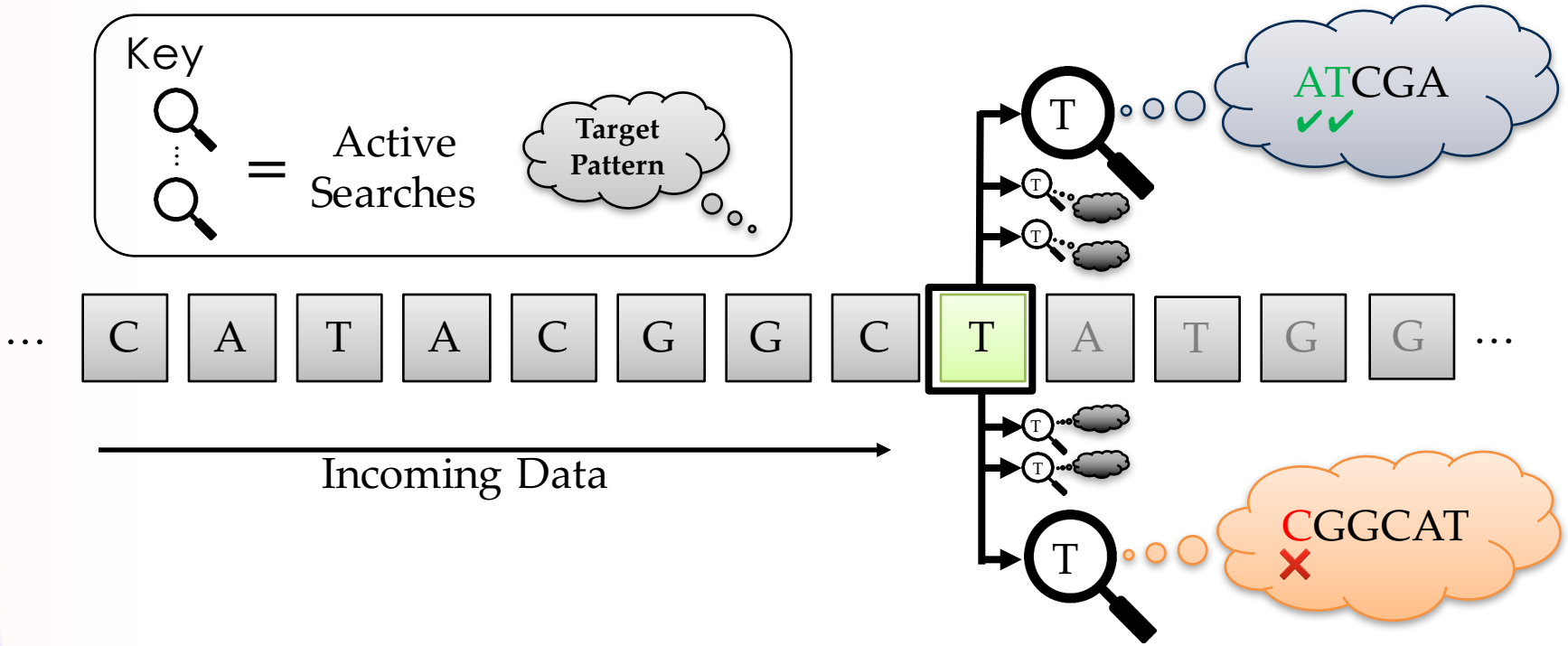
Parallel searches



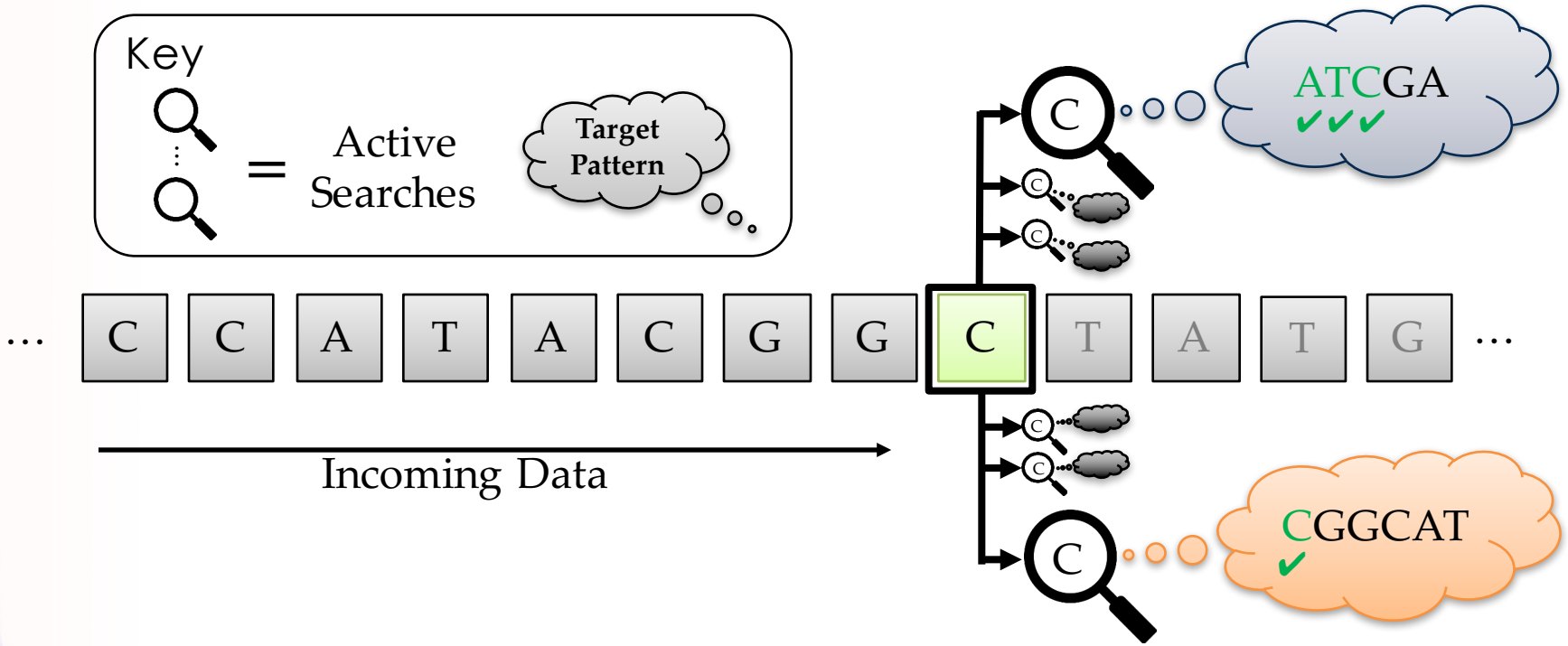
Parallel searches



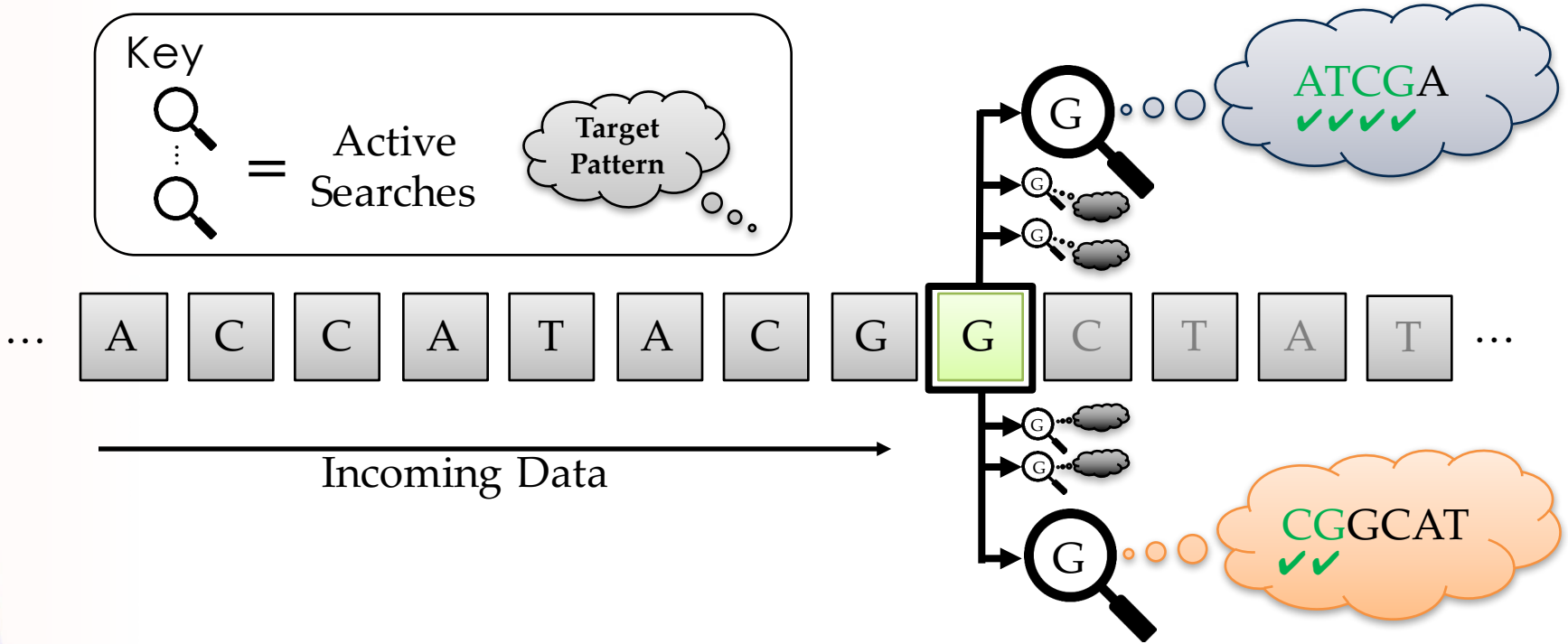
Parallel searches



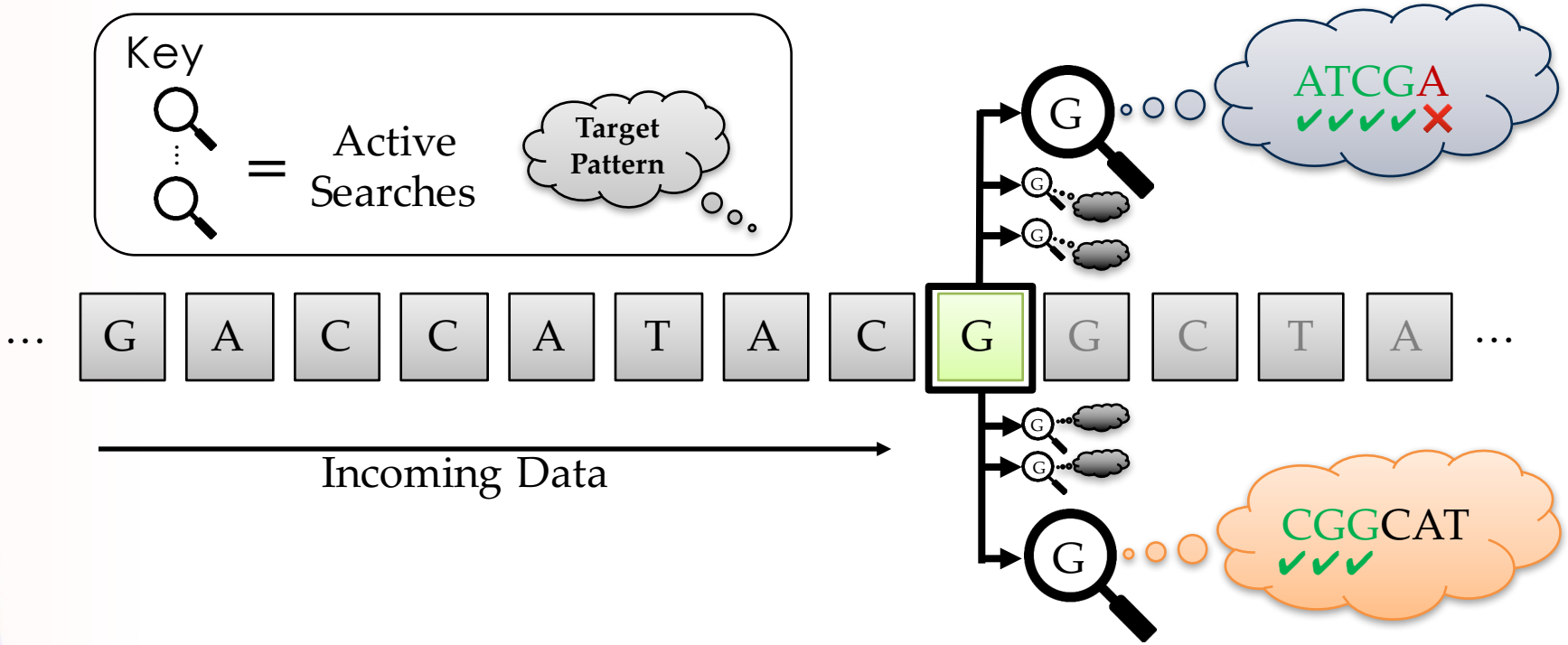
Parallel searches



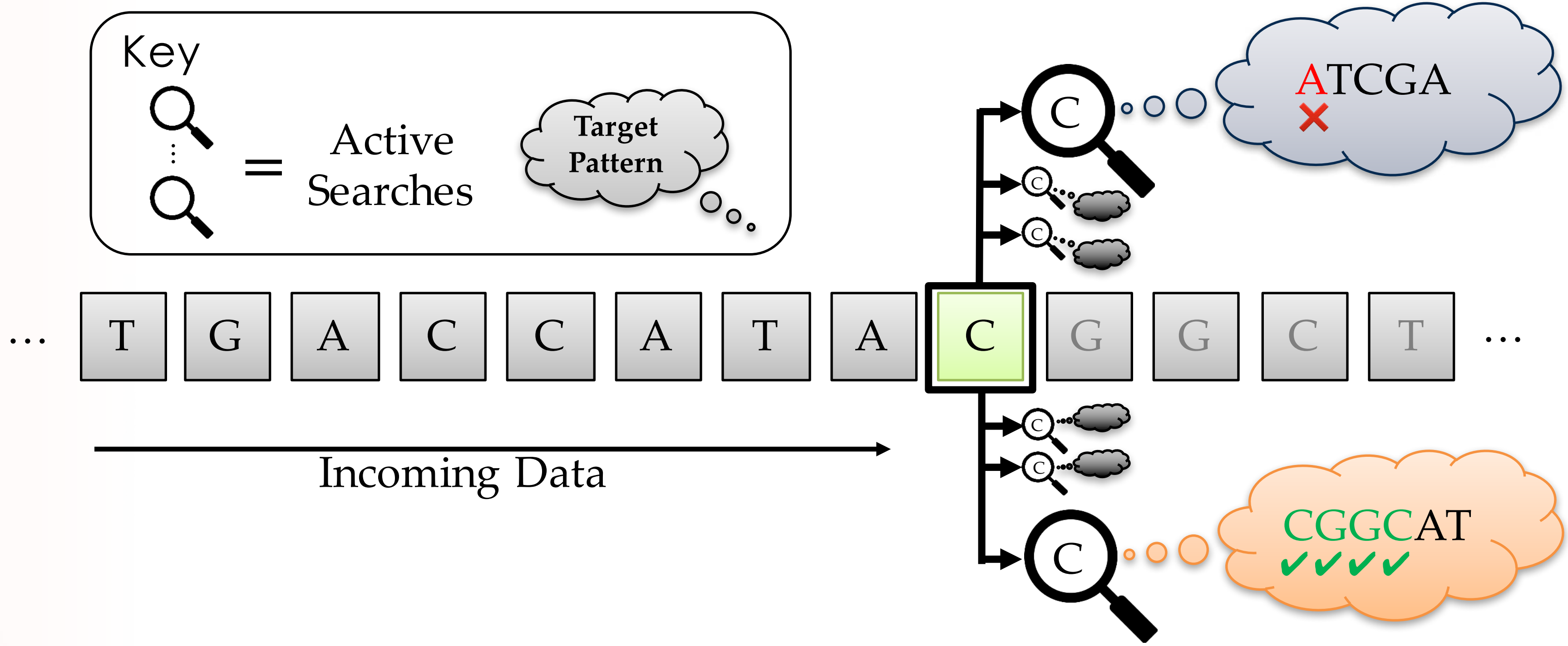
Parallel searches



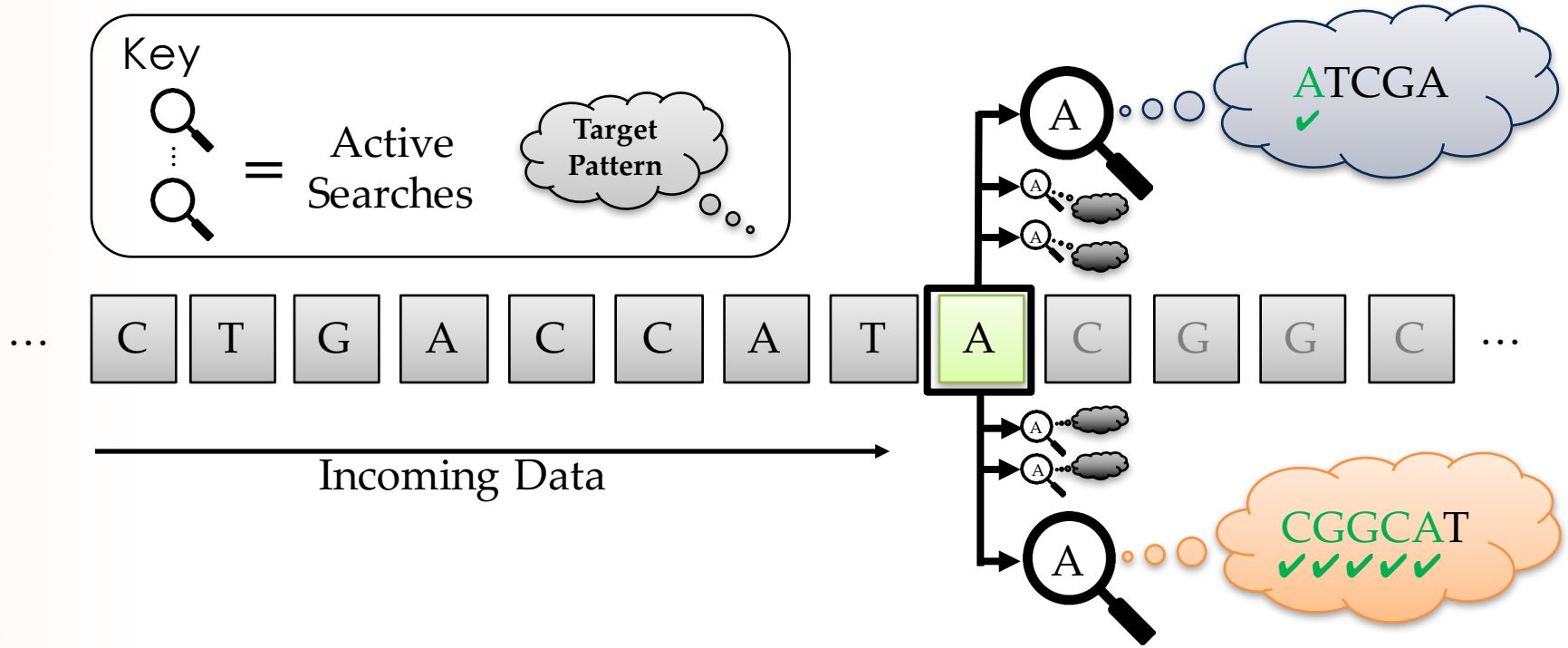
Parallel searches



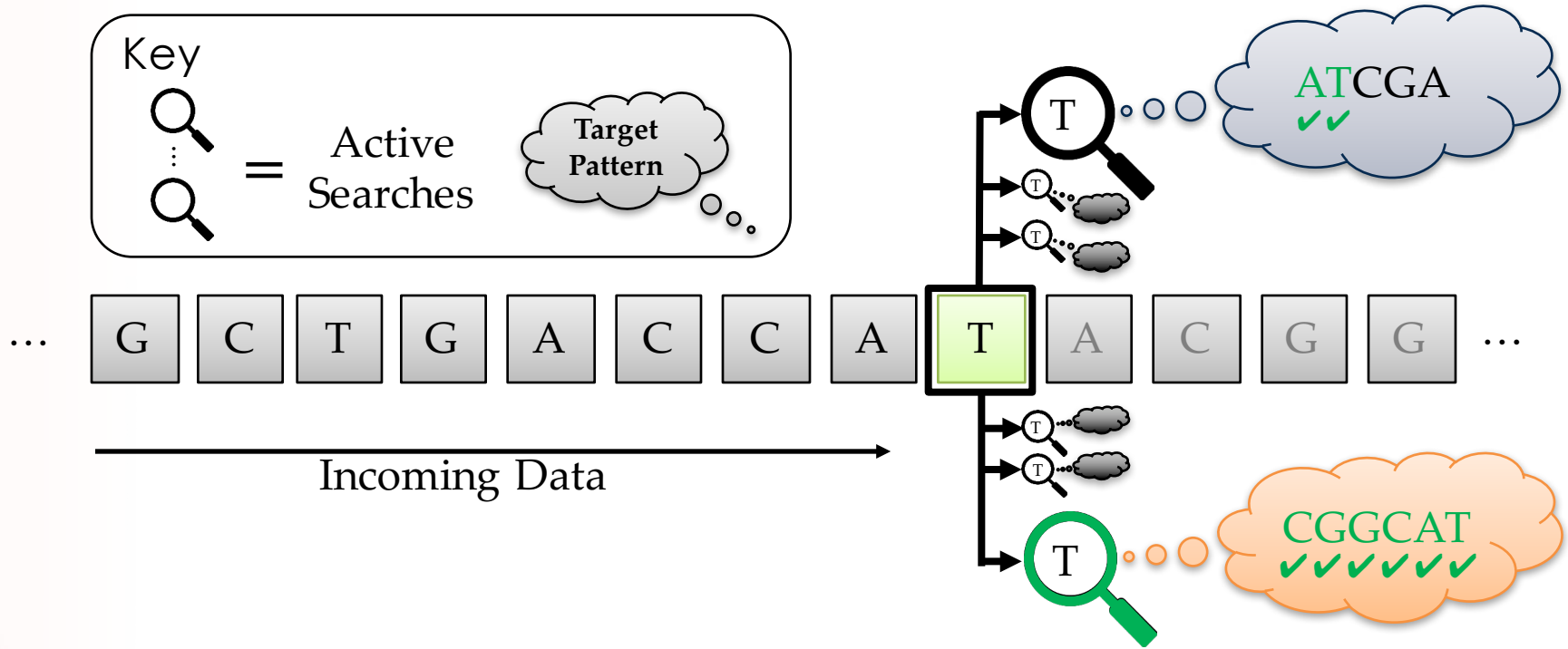
Parallel searches



Parallel searches



Parallel searches



Parallel Searches: Goals

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

Specialized Hardware

RAPID Programming Language

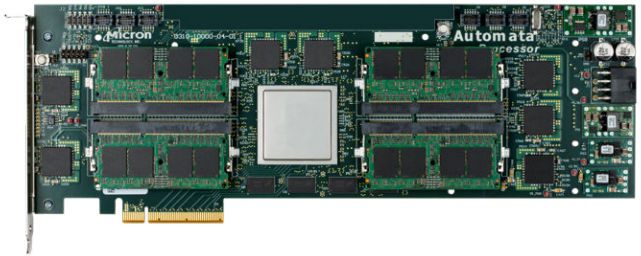
A researcher should spend his or her time designing an algorithm to find the important data, not building a machine that will obey said algorithm.

RAPID Programming

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

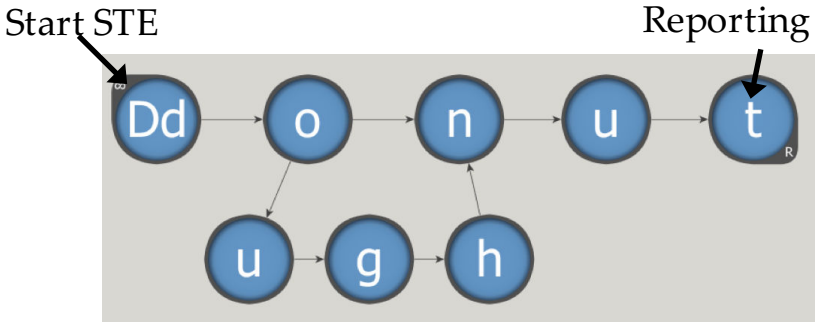
Micron's Automata Processor

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

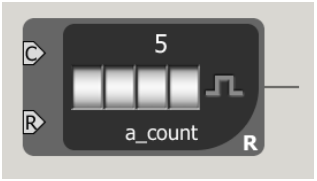


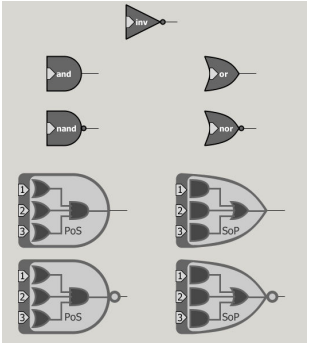
Micron's Automata Processor

- 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



.*[Dd](o|ough)nut





Micron's Automata Processor

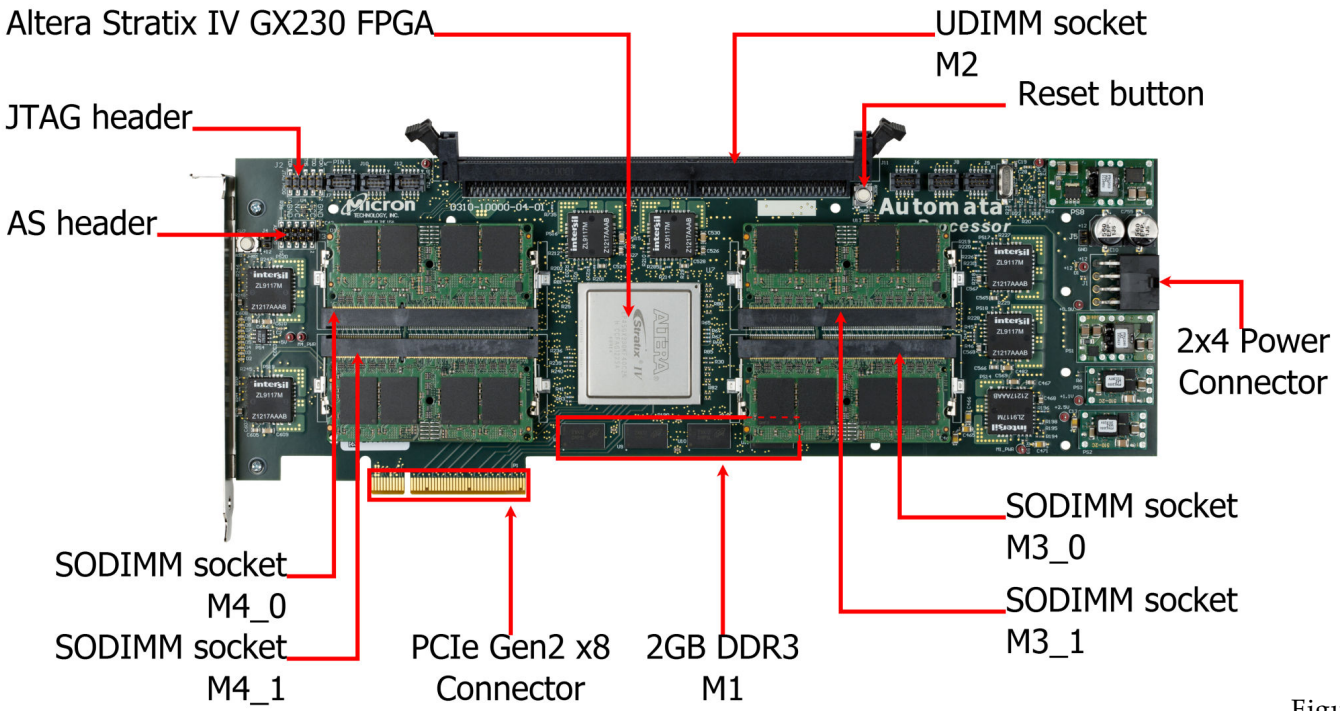
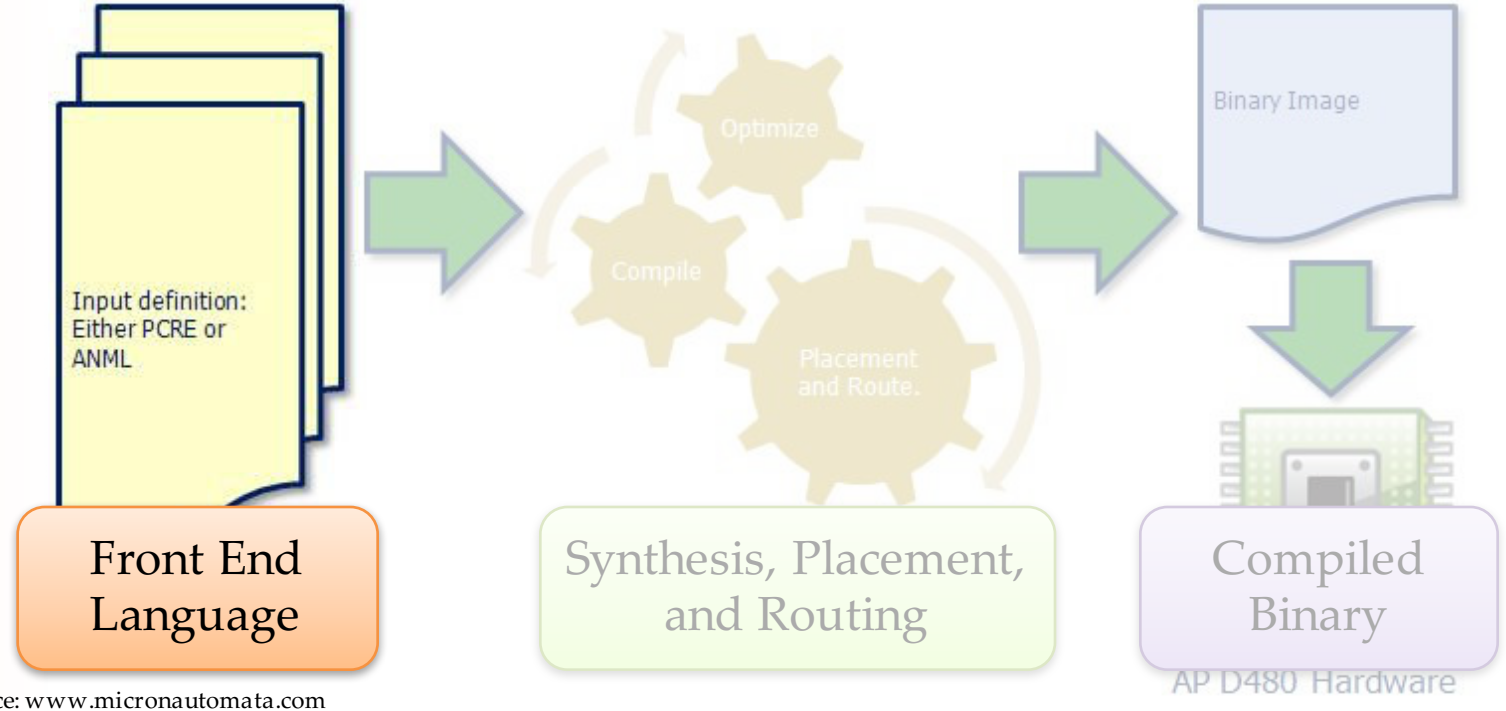


Figure courtesy of Micron

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

RAPID Programming

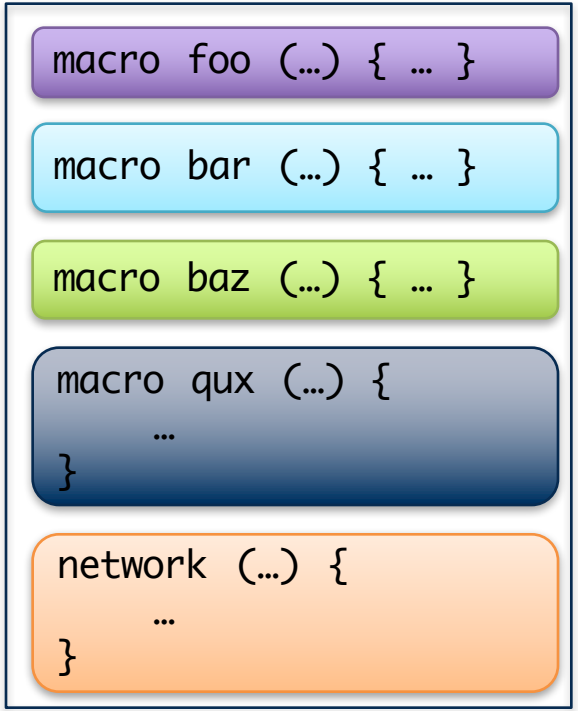
- Pattern-Based Data Analysis
- Automata Processor
- Current Programming Models
- RAPID Language Overview
- AP Code Generation and Optimizations
- Evaluation

RAPID at a Glance

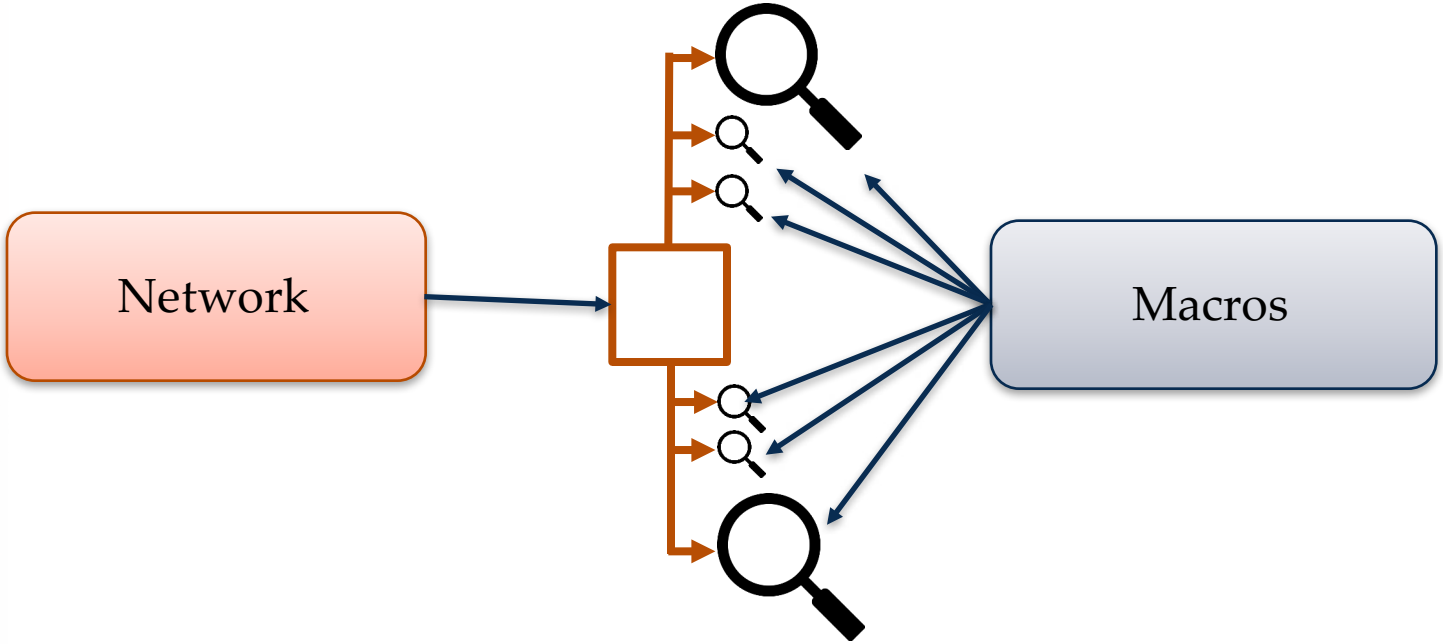
- Provides concise, maintainable, and efficient representations for pattern-identification algorithms
- Conventional, C-style language with domain-specific parallel control structures
- Excels in applications where patterns are best represented as a combination of text and computation
- Compilation strategy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

Program Structure

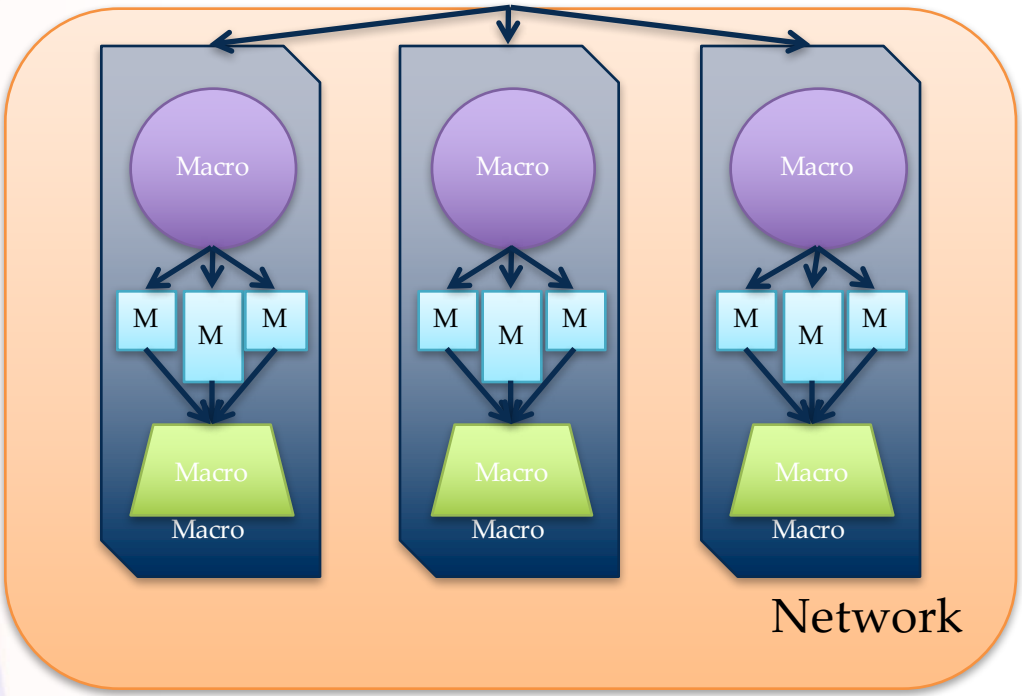
- **Macro**
  - Basic unit of computation
  - Sequential control flow
  - Boolean expressions as statements for terminating threads of computation
- **Network**
  - High-level pattern matching
  - Parallel control flow
  - Parameters to set run-time values



Program Structure



Program Structure




```
macro foo (…) { … }

macro bar (…) { … }

macro baz (…) { … }

macro qux (…) {
    …
}

network ( … ) {
    …
}
```

Data in RAPID

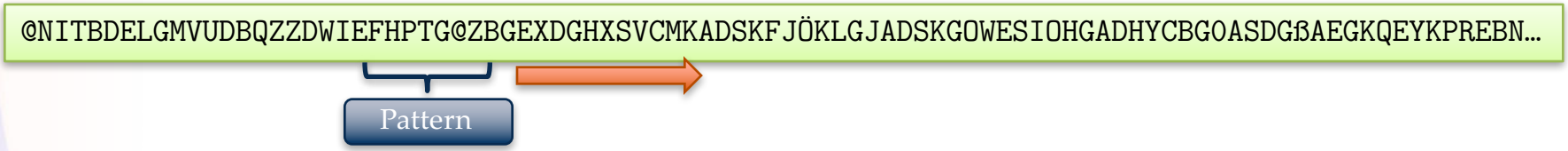
- Input data stream as special function
  - Stream of characters
  - `input()`
    - Calls to `input()` are synchronized across all active macros
    - All active macros receive the same input character

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

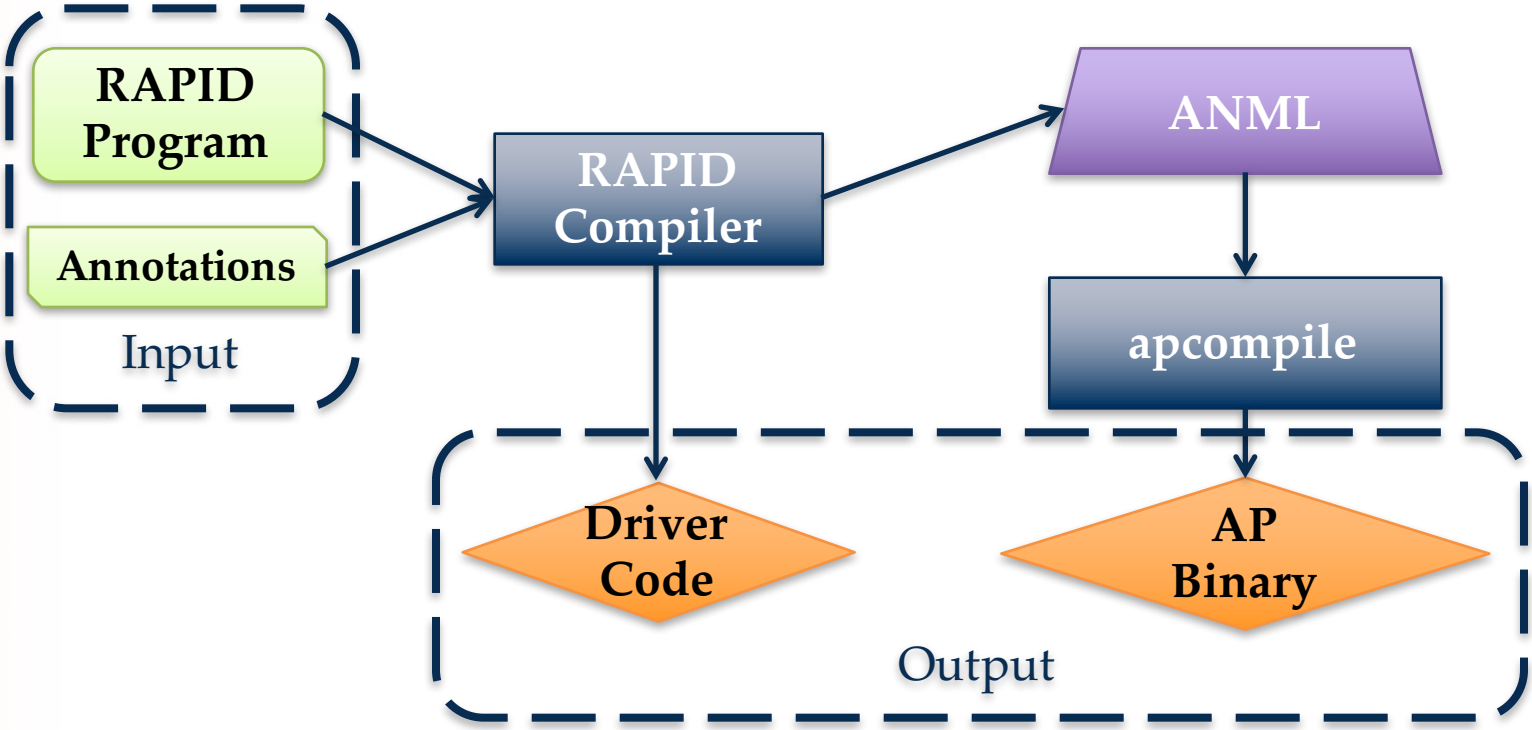
```
macro frequent (String set, Counter cnt) {
        foreach(char c : set) {
            while(input() != c);
        }
        cnt.count();
}

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

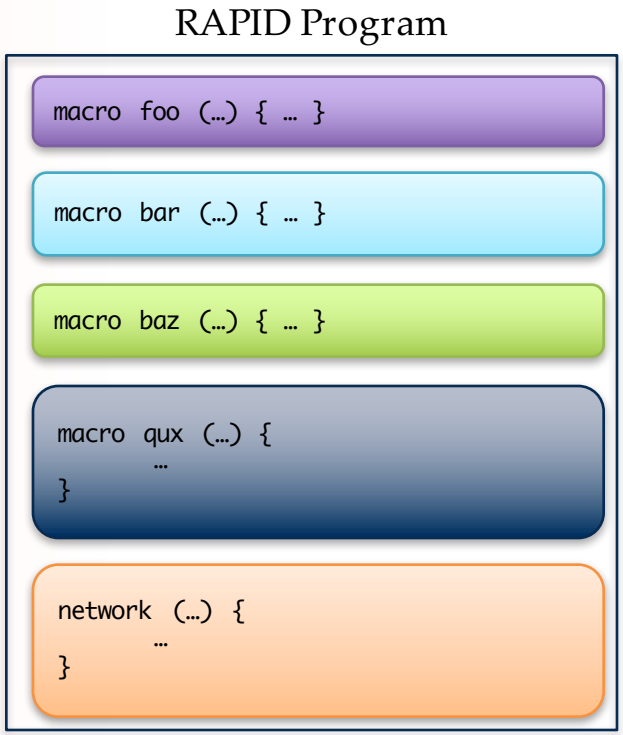
RAPID Programming

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

System Overview



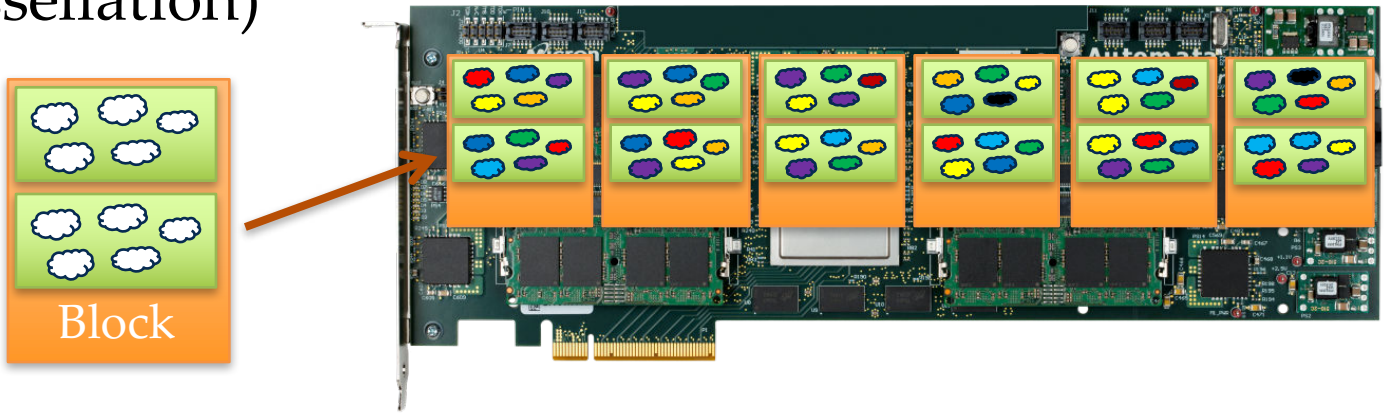
Code Generation



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

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

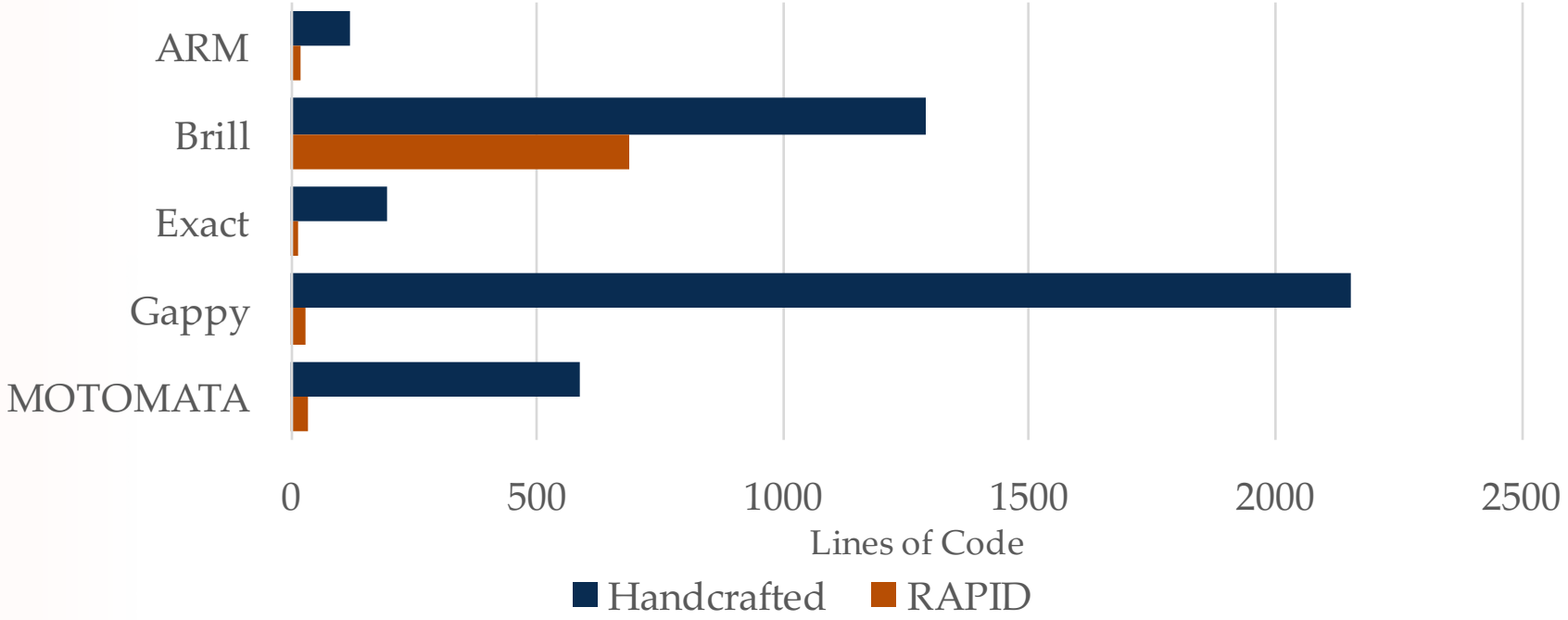
Reminder: Goals

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

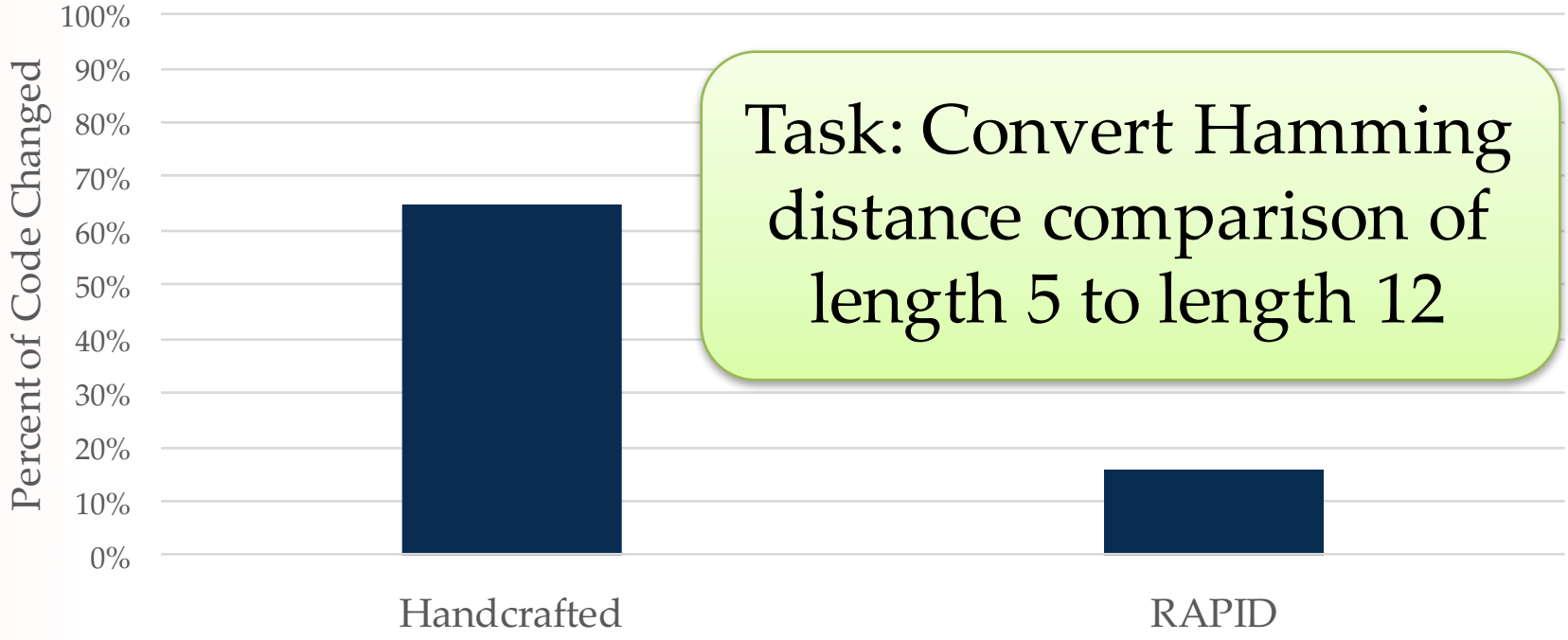
Description of Benchmarks

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

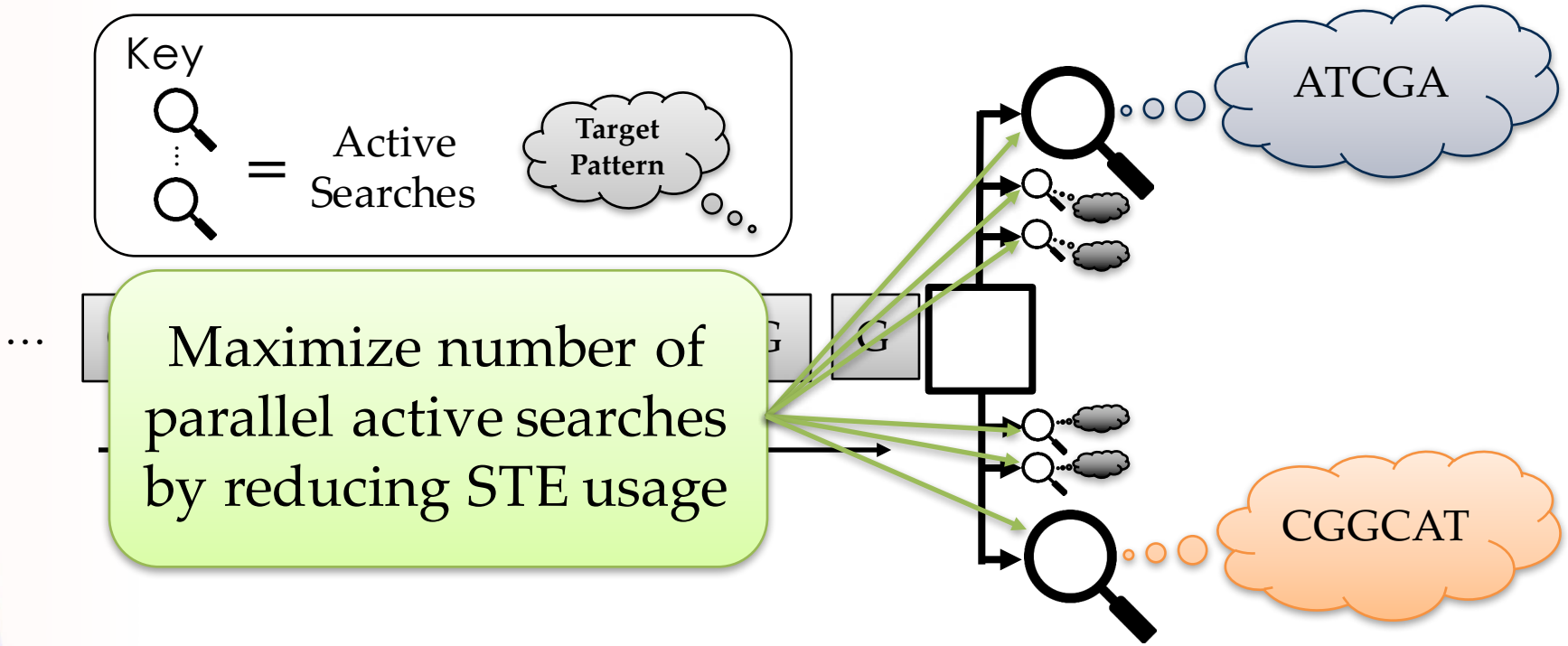
RAPID Lines of Code



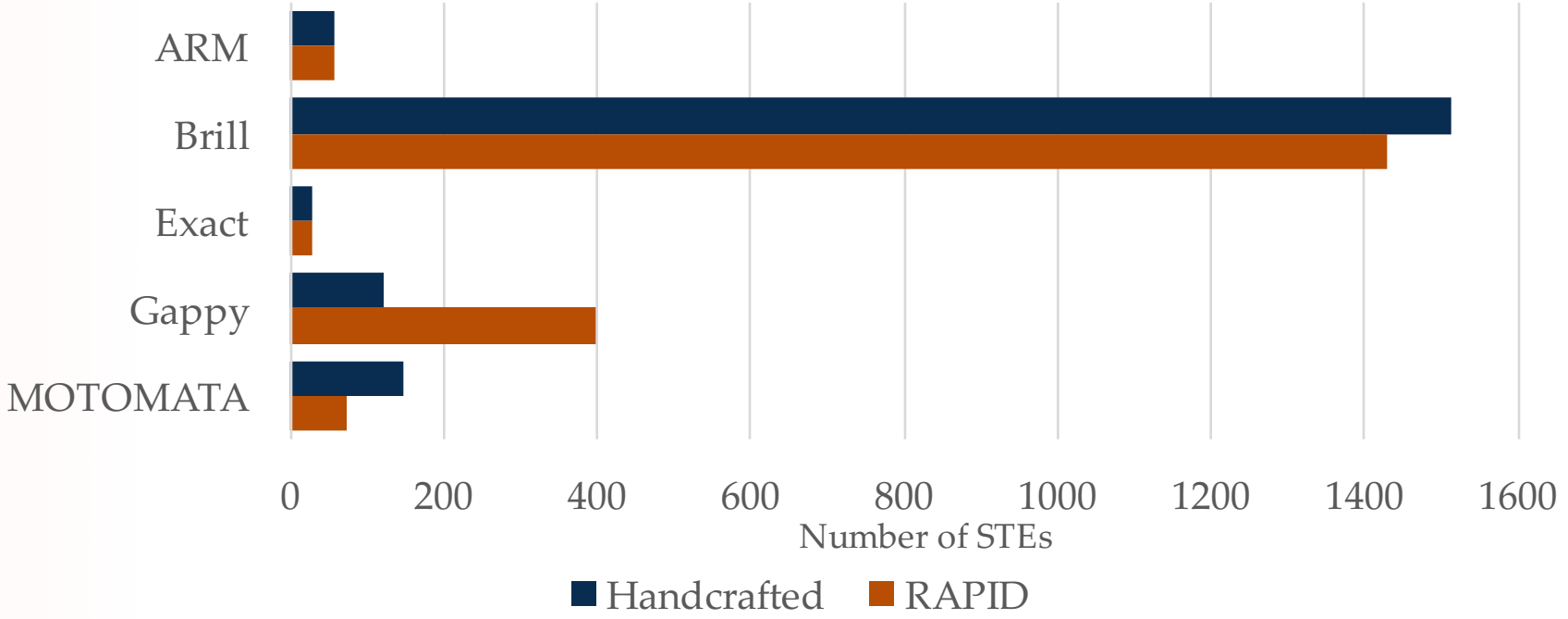
RAPID is Maintainable



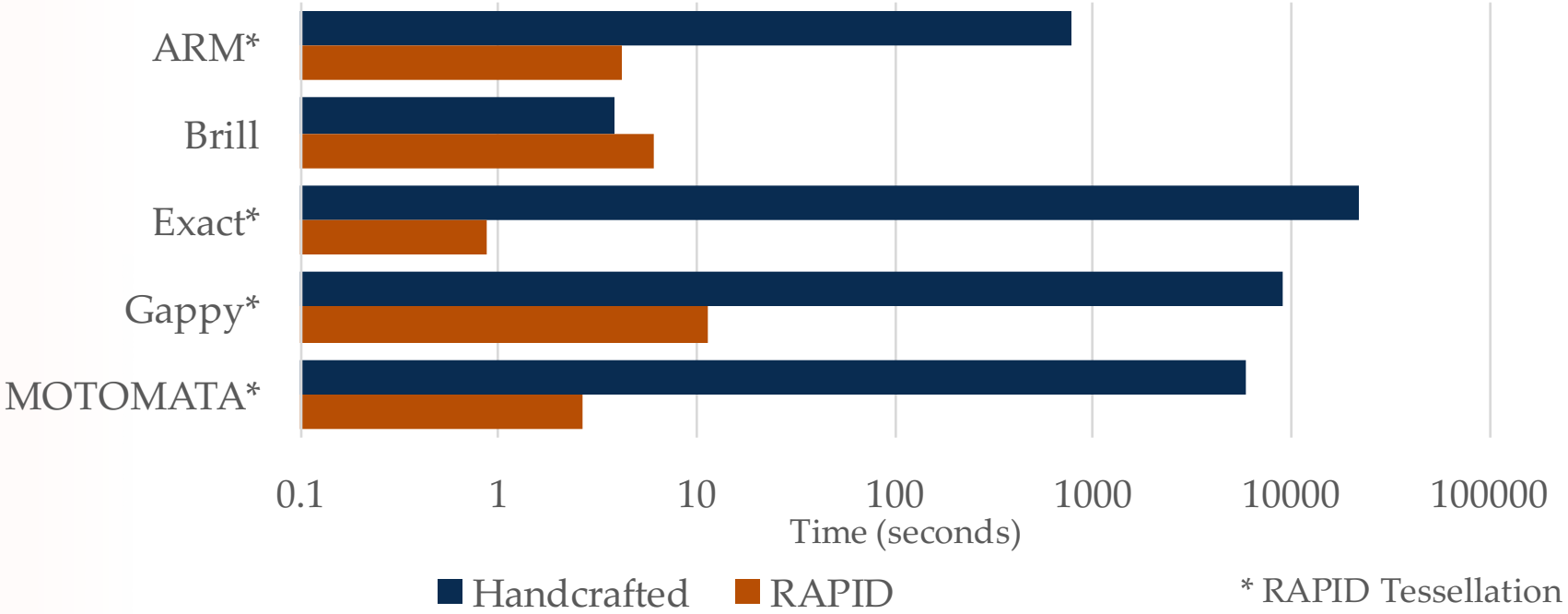
Parallel searches



Generated STEs



Compilation Time

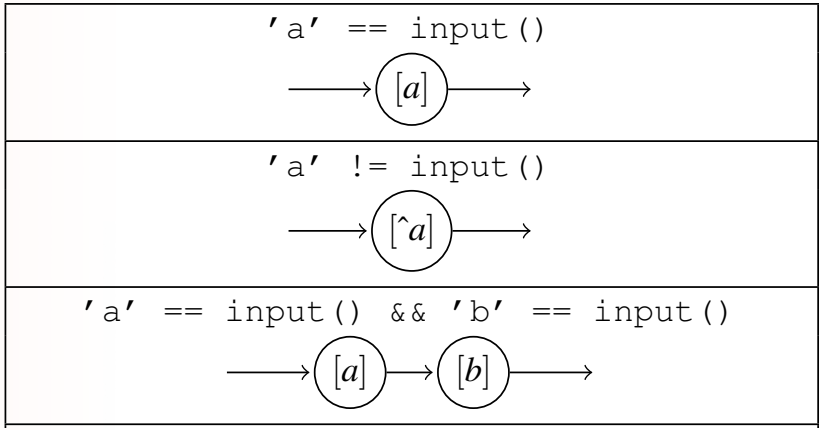


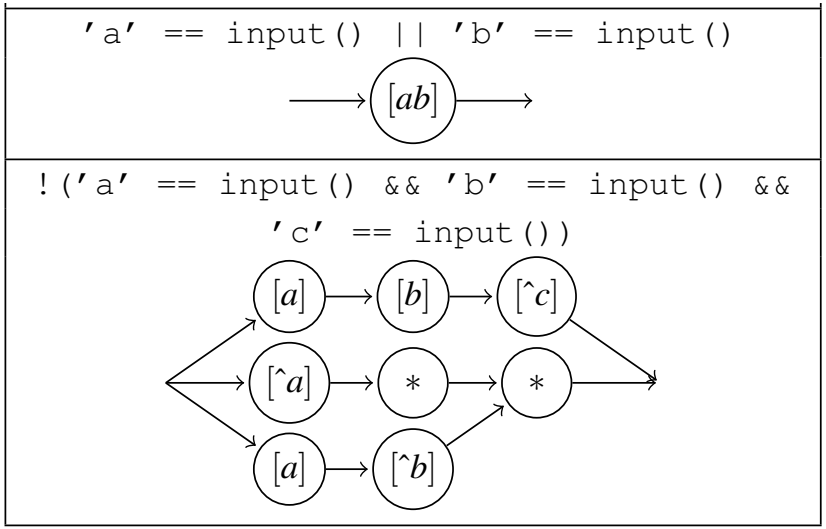
Conclusions

- 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**

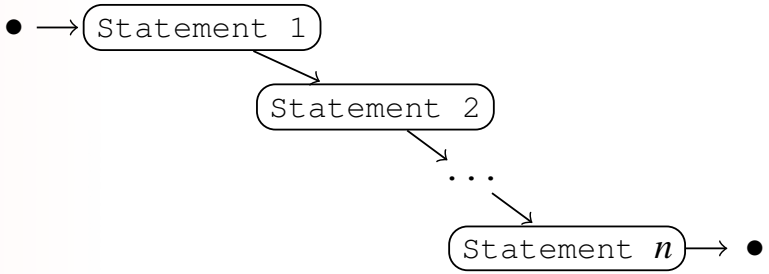
ADDITIONAL SLIDES

Expression Generation

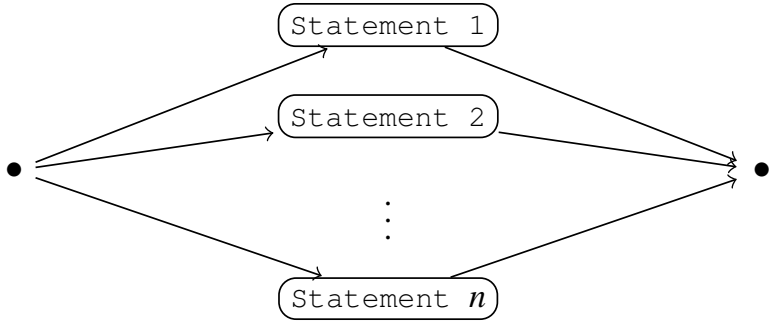




Code Generation

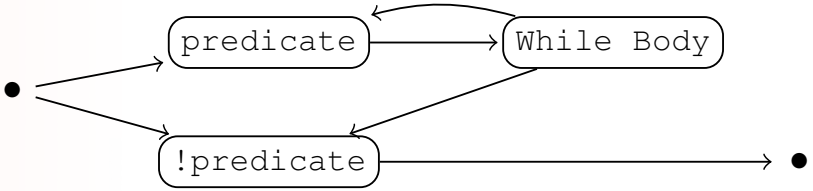


(a) `Foreach` Loops

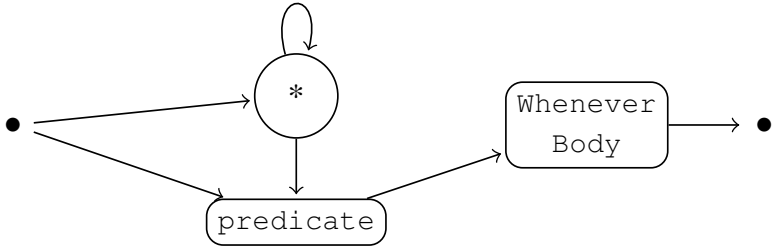


(b) `Either/Orelse` and `Some` statements

Code Generation



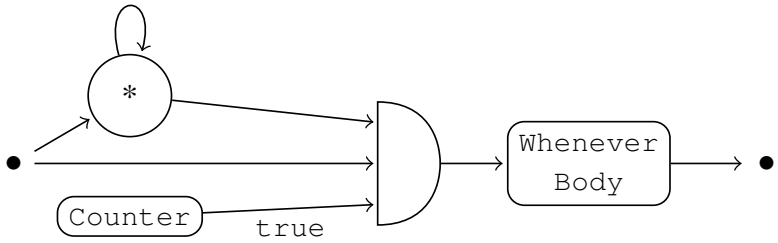
(c) `While` Loops



(d) `Whenever` statement

Counter Generation

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`	



Estimating Runtime

Given n finite automata each consuming m square blocks:

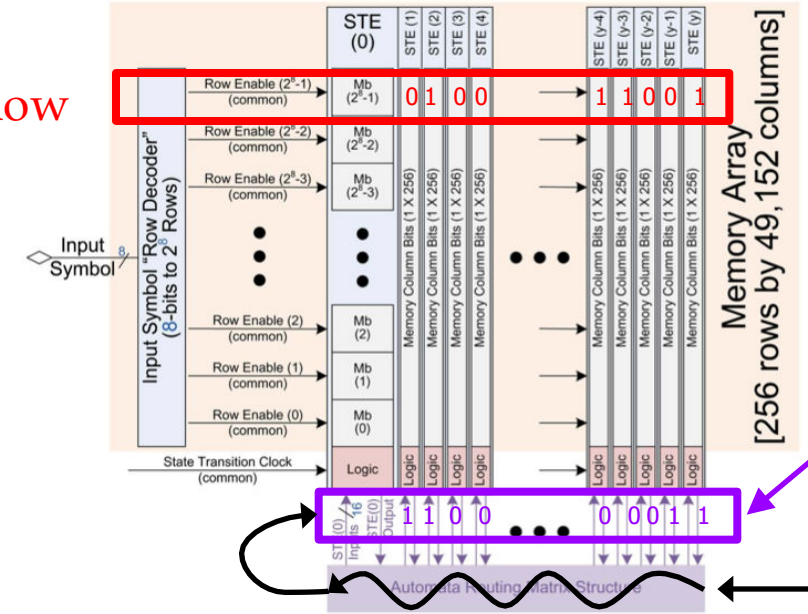
$$runtime \approx \left\lceil \frac{n \cdot m}{6144} \right\rceil \cdot 7.5 \times 10^{-3} seconds$$

(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

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.

RAPID Compilation 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

RAPID Compilation

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.