Just-in-time Length Specialization of Dynamic Vector Code

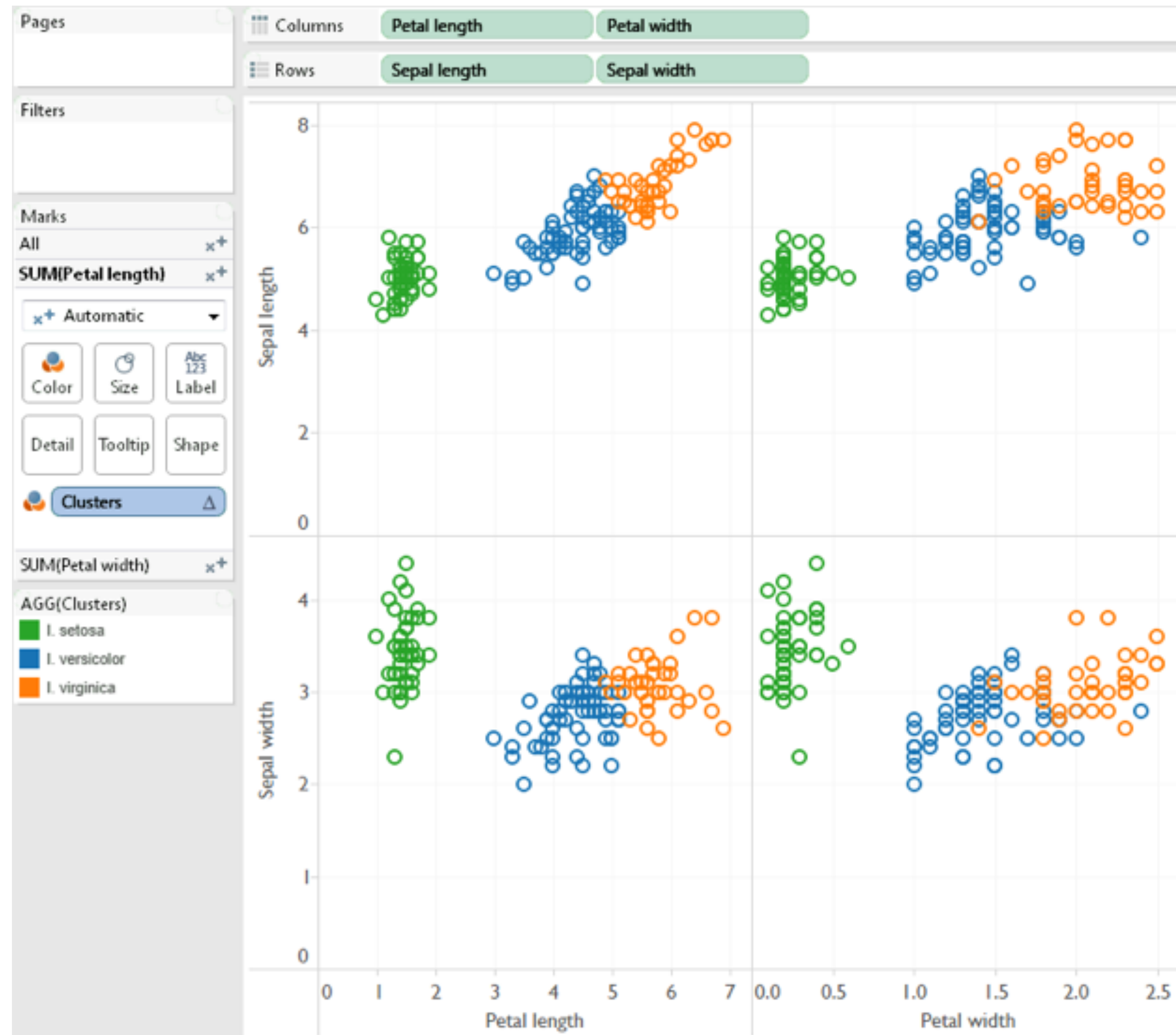
Justin Talbot **Zachary DeVito** **Pat Hanrahan**

Tableau Research Stanford University

(ARRAY 2014)

Tableau

Tableau + R



Riposte

- Bytecode interpreter and tracing JIT compiler for R
- Focused on
    - executing vector code well
    - using parallel hardware
- Written from scratch
  (how fast can it be? don't reason from incremental changes!)
- http://github.com/jtalbot/riposte
- http://purl.stanford.edu/ym439jk6562

What makes R's vectors hard?

They are
semantically poor

How is it used?

- dynamically-allocated array?
- tuple?
- scalar?
- dictionary?
- tree?

What does it imply?

(If I know that a variable is a vector of length 4, what else can I figure out?)

- Usually very little!
- Recycling rule means that almost all vectors conform to each other

Riposte

- Project #1: *Execute long vectors well*
  (large dynamically-allocated arrays)

  - Deferred evaluation approach

  - Operator fusion/merging to eliminate memory bottlenecks

  - Parallelize execution of fused operators

- But…

Riposte

- Project #2: *Execute short vectors well*
  (scalars, tuples, short dynamically-allocated arrays)
    - Hot-loop just-in-time (JIT) compilation
    - (Partial) length specialization
    - Optimize based on lengths

Hot-loop JIT

- Hypothesis: if code has scalars or short vectors, computation time must be dominated by loops.
- Interpreter watches for expensive loops.
- When it finds one, compile machine code for loop, *make assumptions that lead to optimizations (specialization)*
- Guard against changes to assumptions

Hot-loop JIT

- Specialization
  - Assumptions should lead to *big optimization wins* (frequency * performance improvement)
  - Assumptions should be *predictable* (to amortize overhead)

Specialization

- *Type* specialization explored in other dynamic languages (Javascript, etc.)
- *Length* specialization is interesting in R
    - Eliminate recycling overhead
    - Store vector in register/stack instead of heap
    - Length-based optimizations (fusion, etc.)

Which length specializations make sense?

(big win + predictable)

Length specializations?

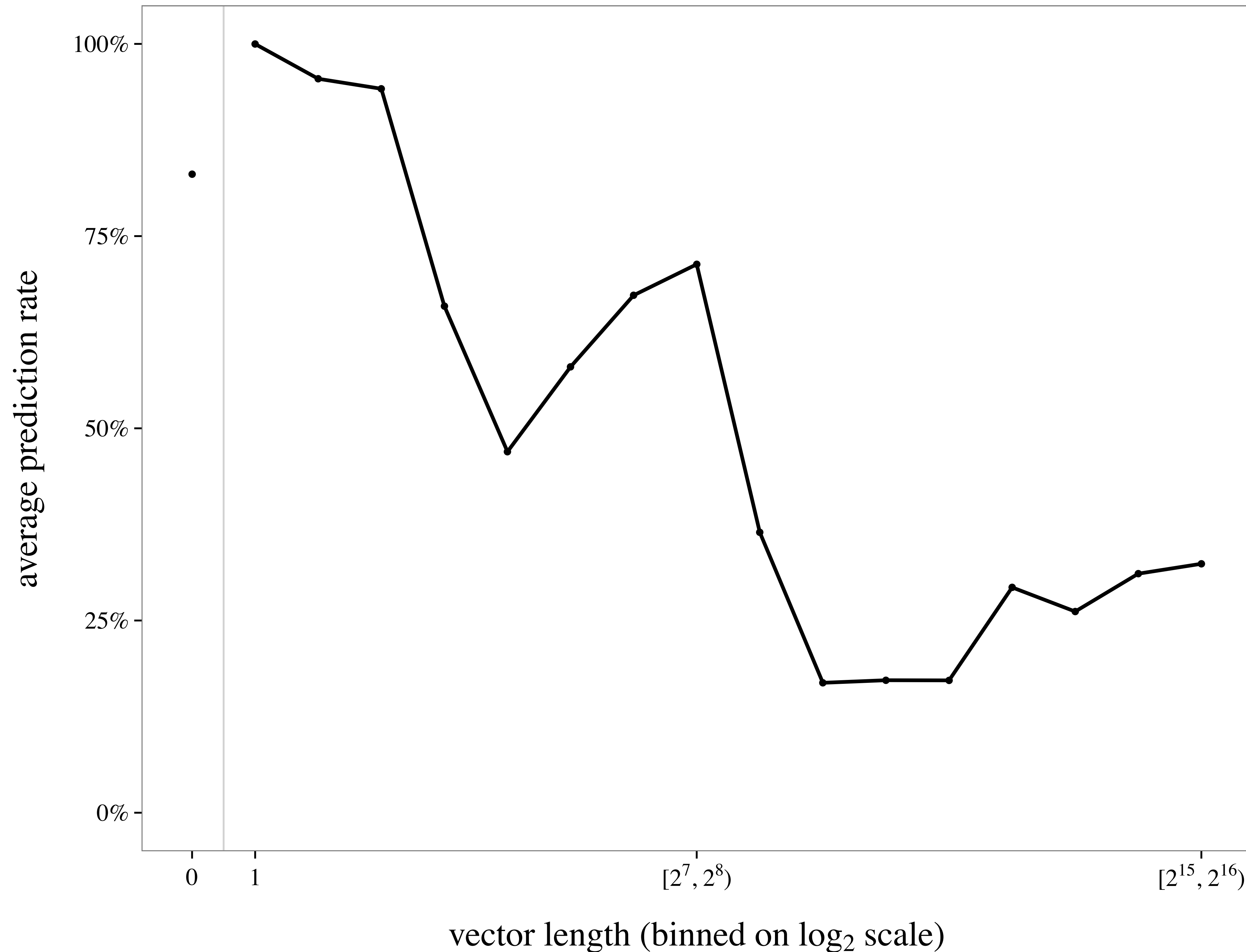
- Instrumented GNU R
- Recorded operand lengths of binary arithmetic operators
- Ran 200 vignettes, covering wide range of R application areas

Recycling rule?

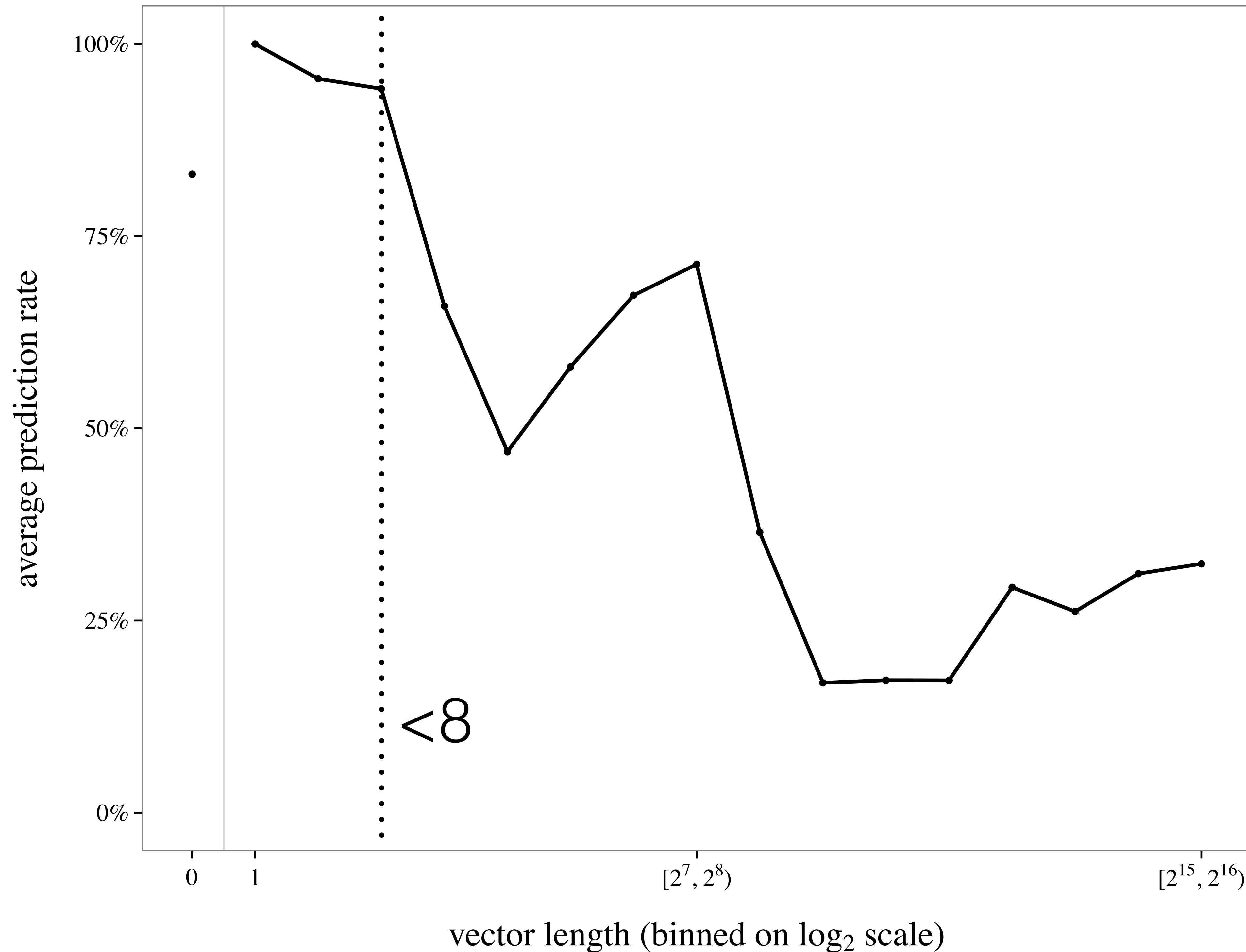
- In 92% of calls, operands are the same length
  - ➡ Recycling overhead is frequently unnecessary
- Recycling is well predicted
  - Same lengths: 99.998%
  - Different lengths: 99.98%
  - ➡ Specialized code has a high probability of being reused

Predictable lengths?

Predictable lengths?



Predictable lengths?



Our strategy

Partial length specialization

1. Record loop using recycle instructions + abstract lengths

2. Eliminate *some* recycle instructions + introduce guards

    - Heuristic: Only specialize if the input lengths were equal while tracing and if both are loop carried or if both aren't

3. Specialize *some* abstract lengths to concrete lengths + introduce guards

    - Heuristic: Only specialize vectors with non-loop carried lengths <= 4

Length-based optimizations

- Operator fusion
  (can’t have intervening recycle operations)

- Vector “register allocation”

  - SSE registers
    (needs concrete lengths)

  - Shared stack/heap locations / eliminate copies
    (needs same lengths)

Evaluation

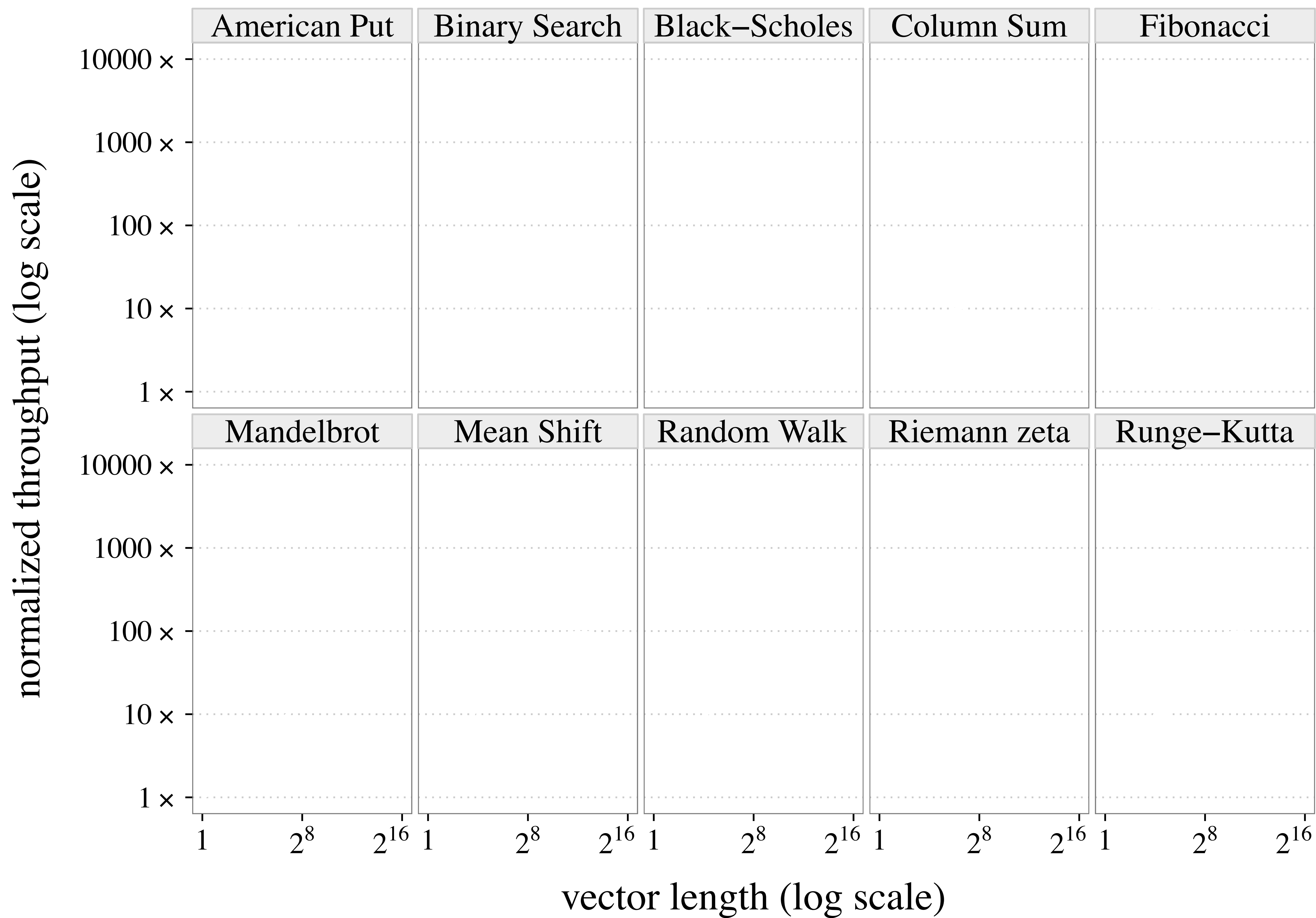
Evaluation

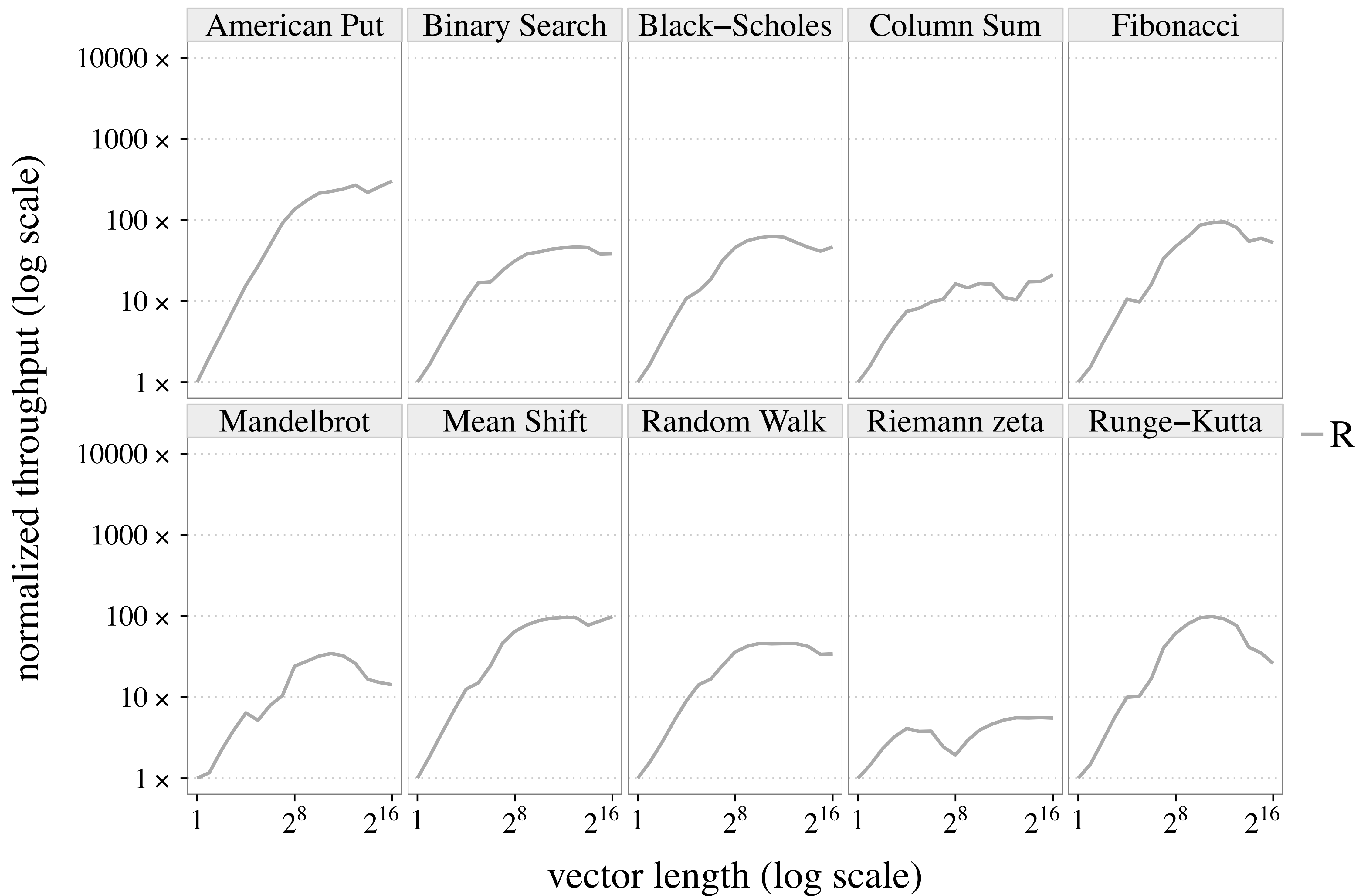
- **Can we run vectorized code efficiently across a wide range of vector lengths?**

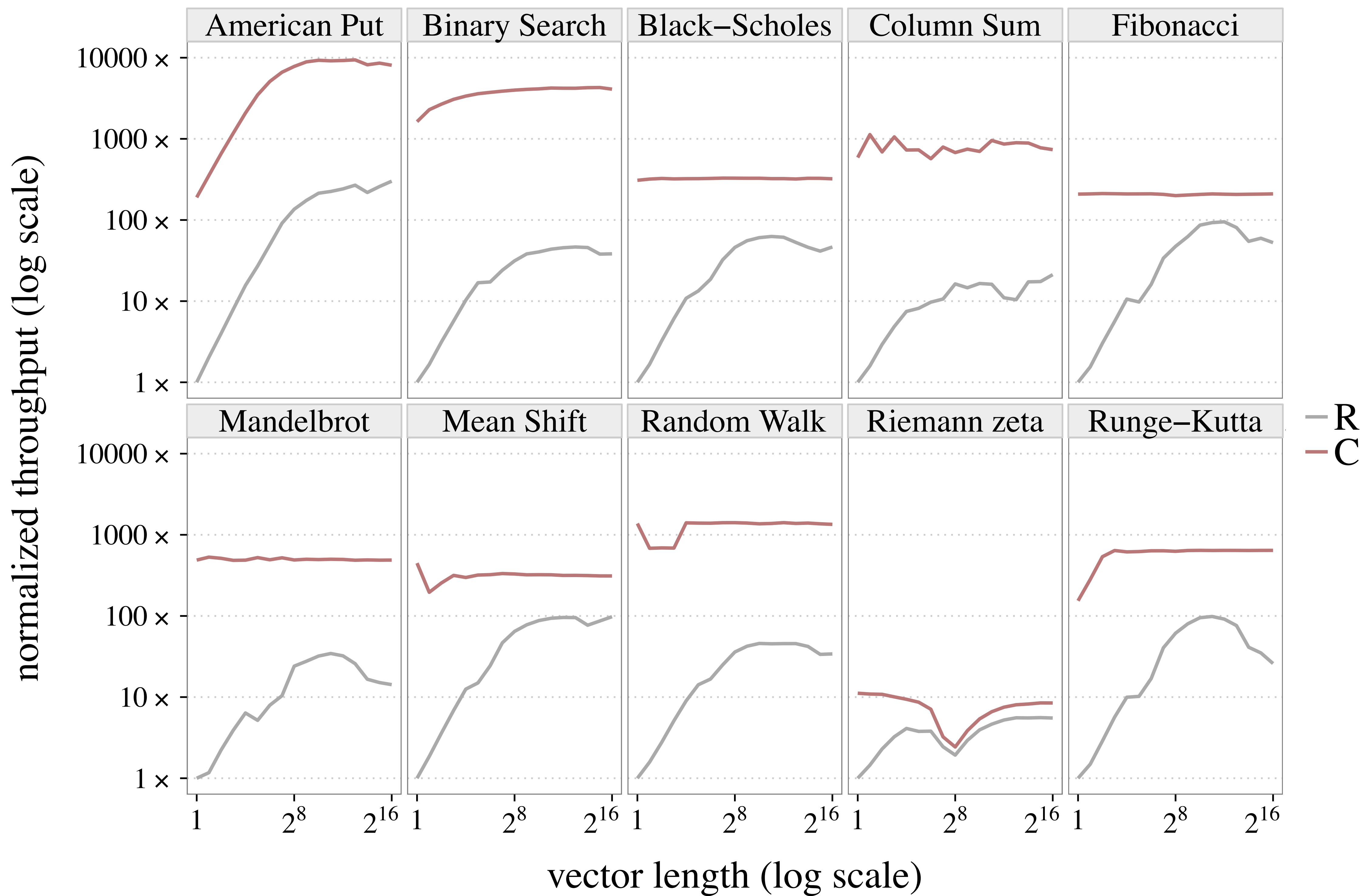
  - 10 workloads, written in idiomatic R vectorized style so we can vary length of input vectors

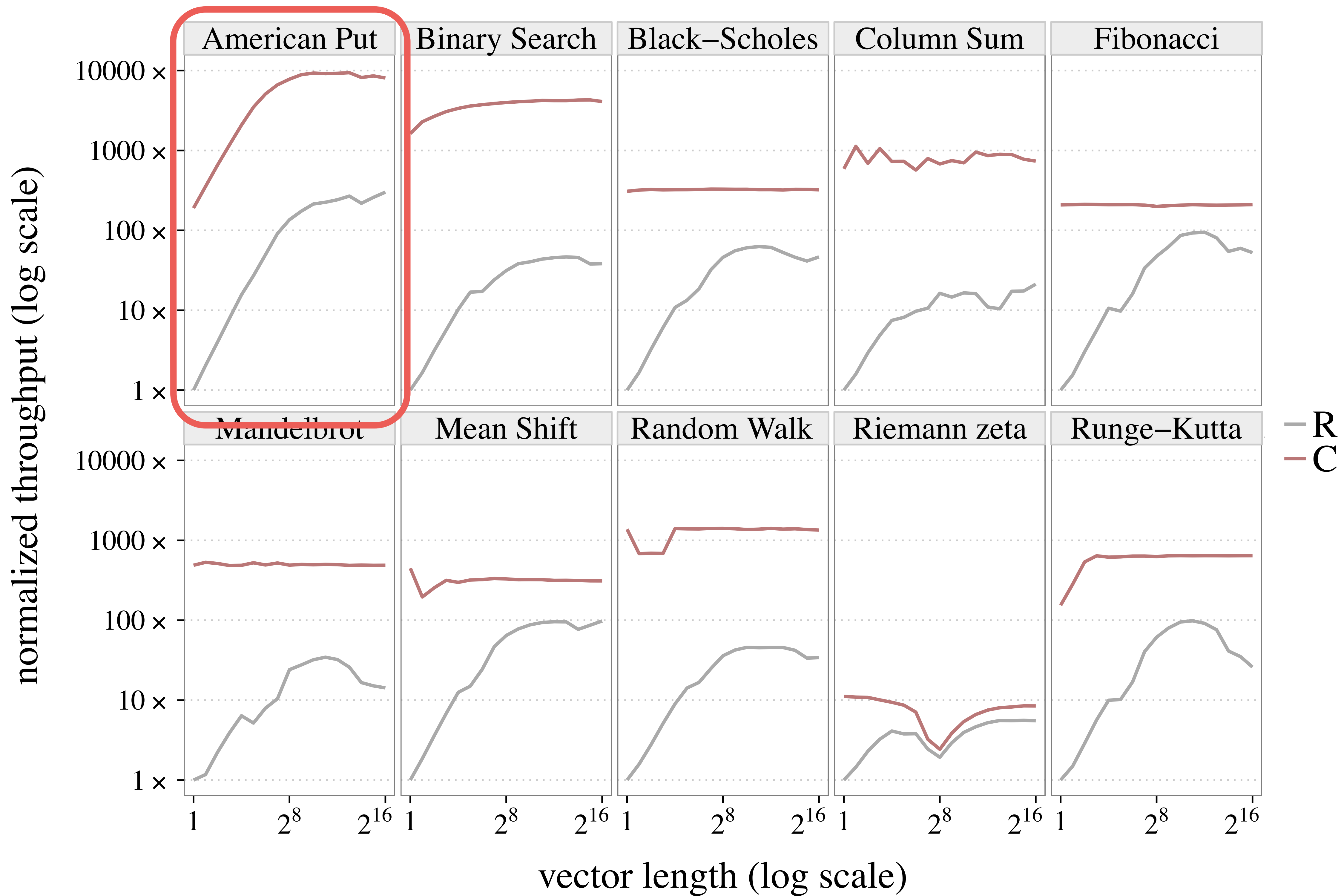
  - Compare to GNU R bytecode interpreter & C (clang 3.1 -O3 + autovectorization)

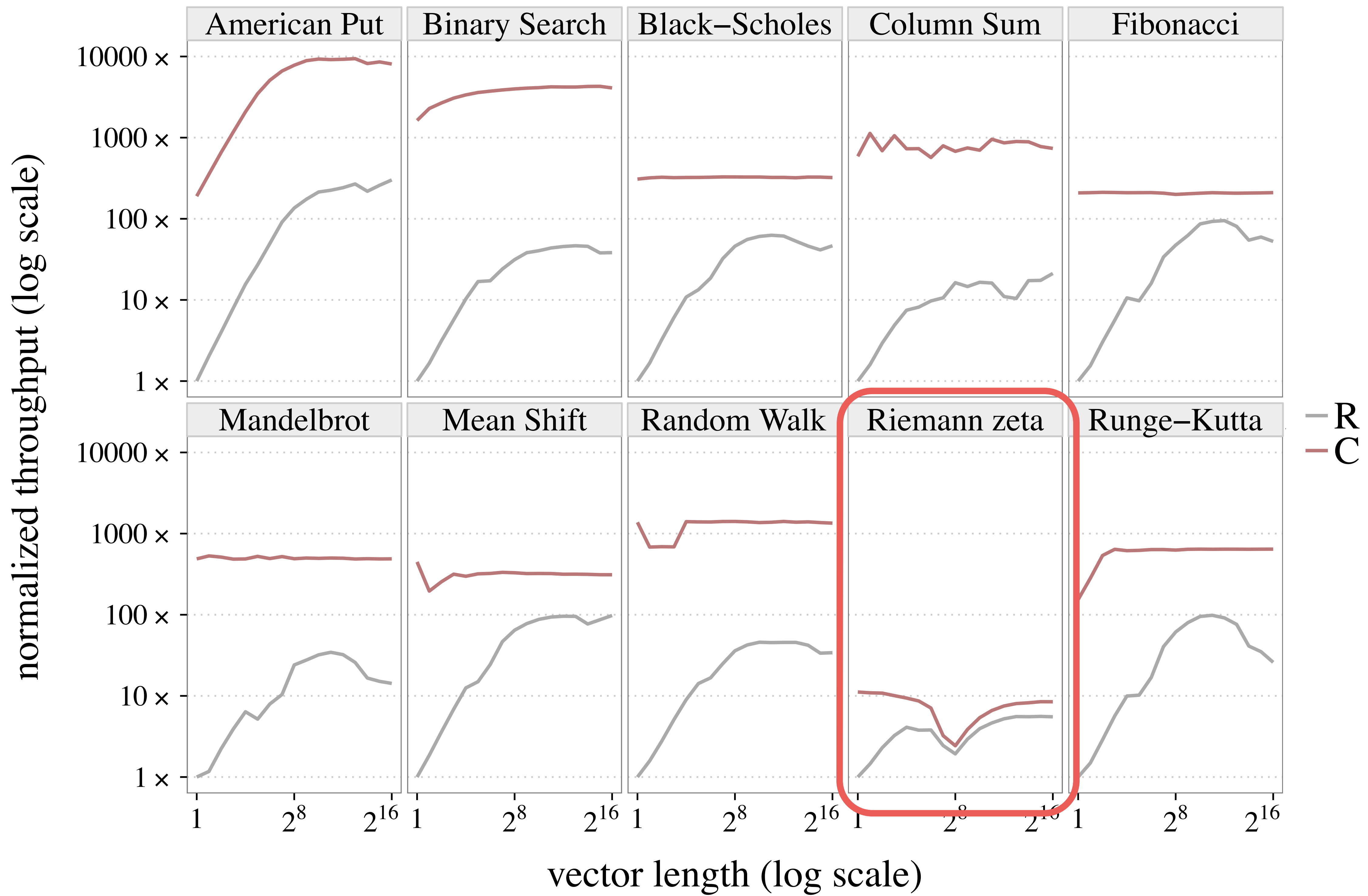
  - Measure just execution time

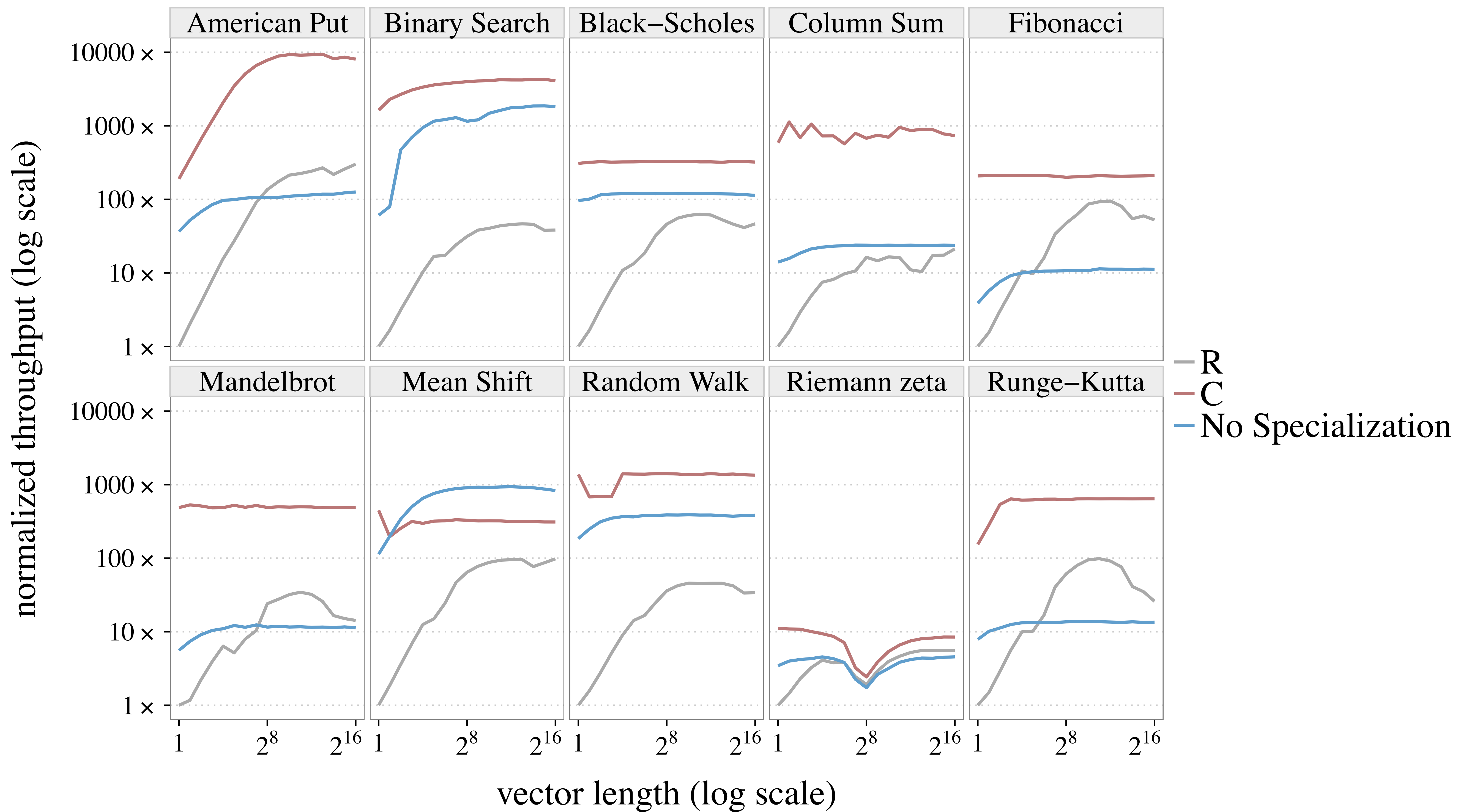

American Put
Binary Search
Black−Scholes
Column Sum
Fibonacci
Mandelbrot
Mean Shift
Random Walk
Riemann zeta
Runge−Kutta
10000 ×
1000 ×
100 ×
10 ×
1 ×
normalized throughput (log scale)
1
2^8
2^16
vector length (log scale)

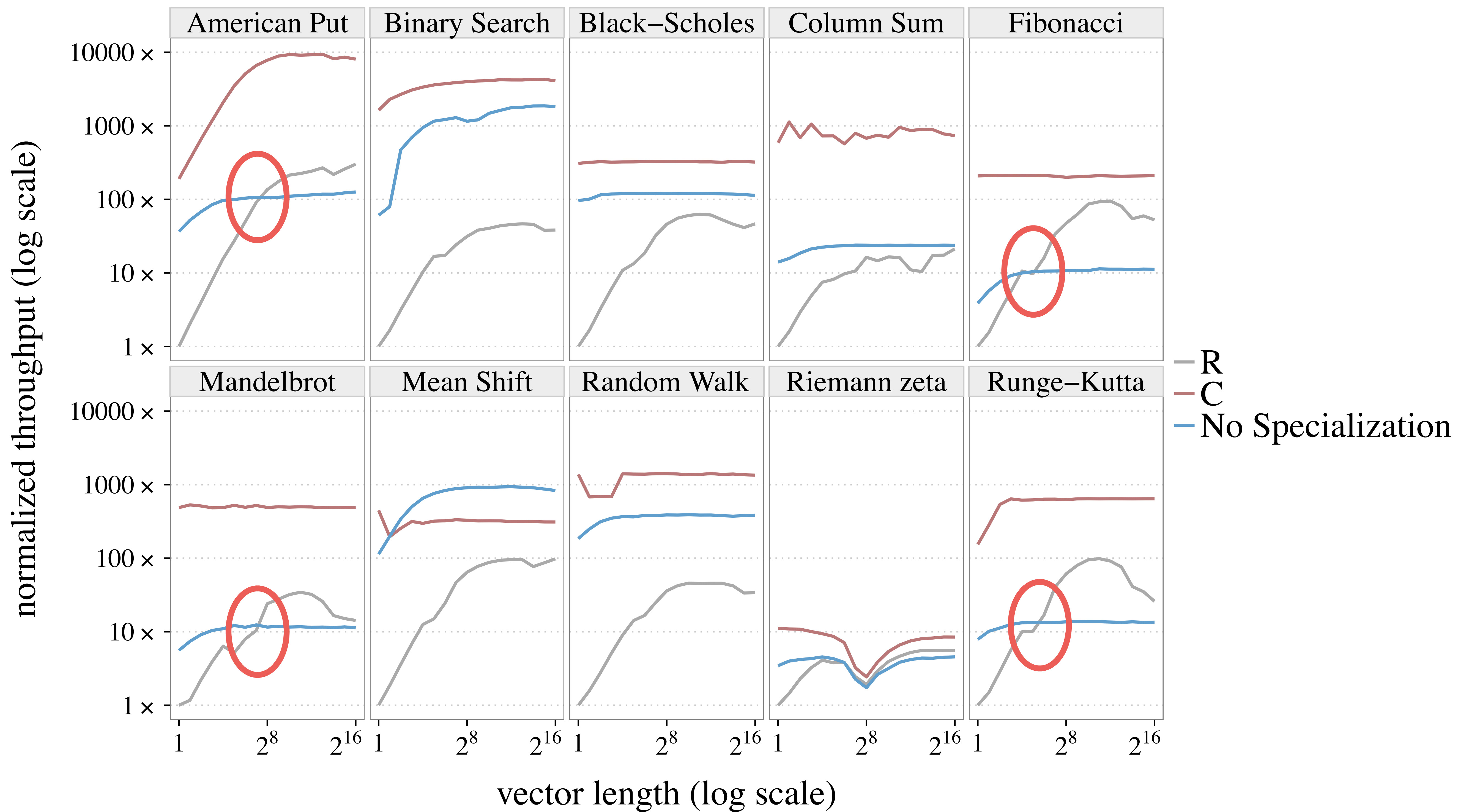

American Put
Binary Search
Black−Scholes
Column Sum
Fibonacci
Mandelbrot
Mean Shift
Random Walk
Riemann zeta
Runge−Kutta
10000 ×
1000 ×
100 ×
10 ×
1 ×
1
2^8
2^{16}
R
normalized throughput (log scale)
vector length (log scale)

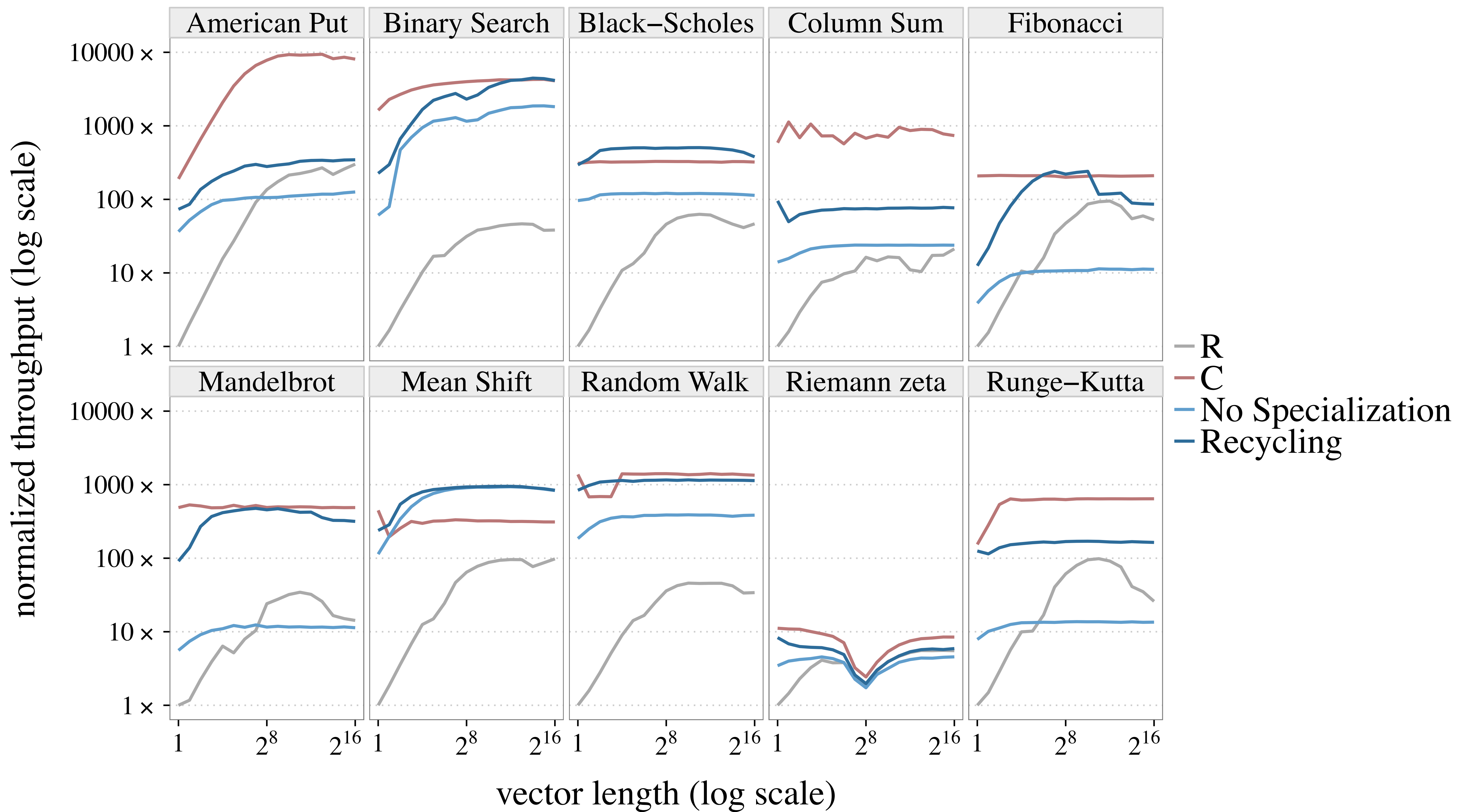

American Put
Binary Search
Black−Scholes
Column Sum
Fibonacci
Mandelbrot
Mean Shift
Random Walk
Riemann zeta
Runge−Kutta
normalized throughput (log scale)
10000 ×
1000 ×
100 ×
10 ×
1 ×
vector length (log scale)
1
2^8
2^16
R
C

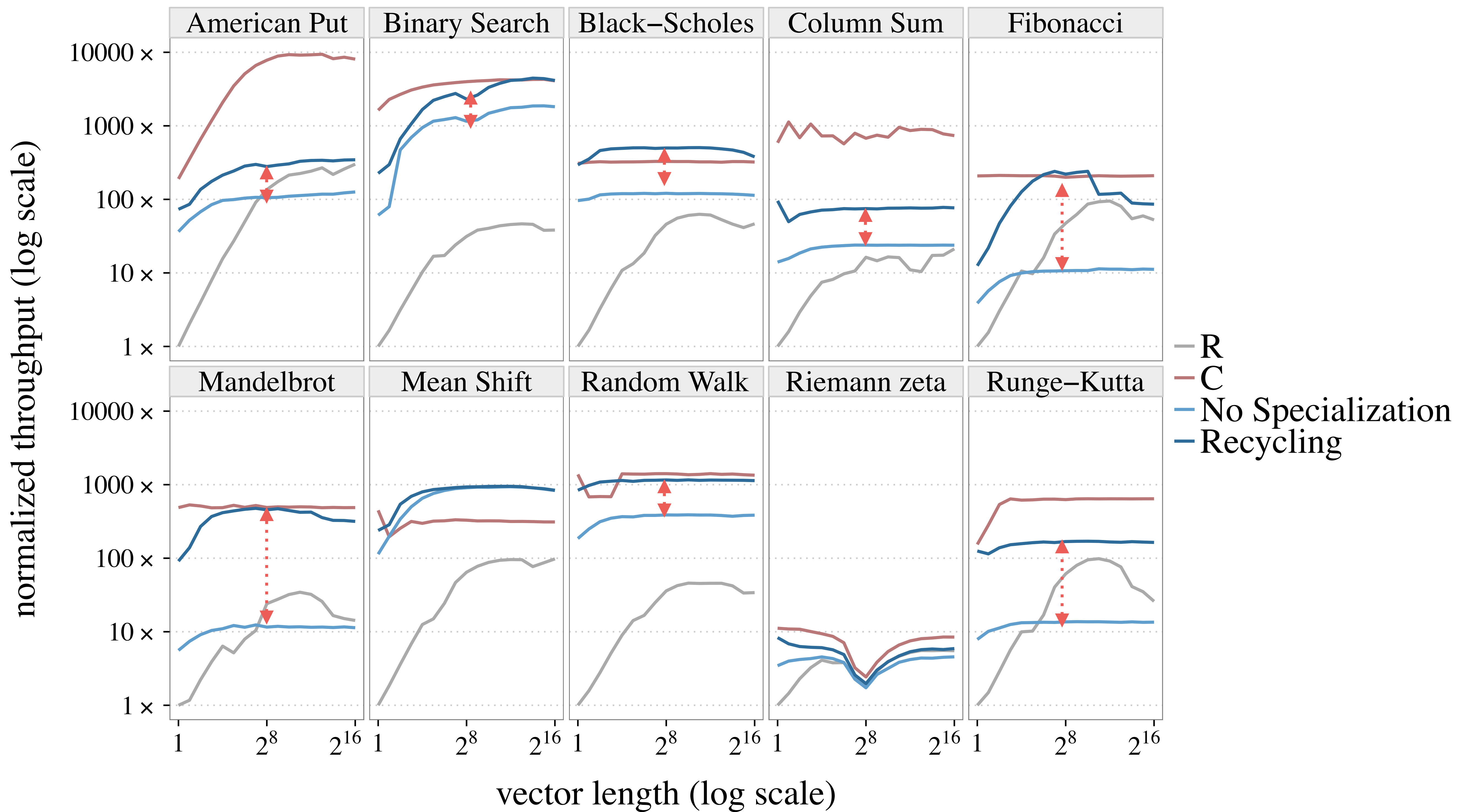

American Put
Binary Search
Black–Scholes
Column Sum
Fibonacci
Mandelbrot
Mean Shift
Random Walk
Riemann zeta
Runge–Kutta
10000 ×
1000 ×
100 ×
10 ×
1 ×
normalized throughput (log scale)
1
2^8
2^16
vector length (log scale)
R
C

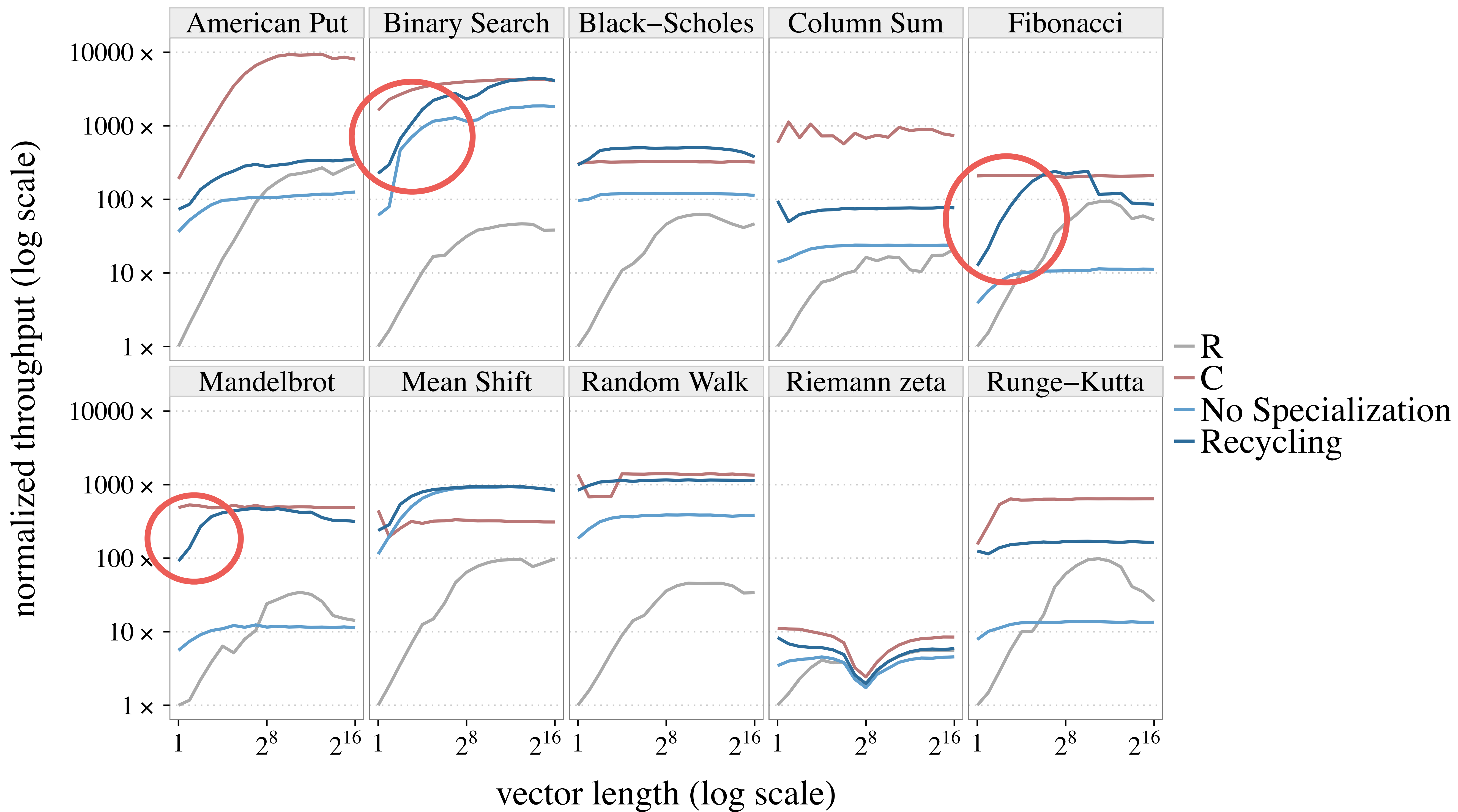

American Put
Binary Search
Black−Scholes
Column Sum
Fibonacci
Mandelbrot
Mean Shift
Random Walk
Riemann zeta
Runge−Kutta
10000 ×
1000 ×
100 ×
10 ×
1 ×
1
2^8
2^16
R
C
normalized throughput (log scale)
vector length (log scale)

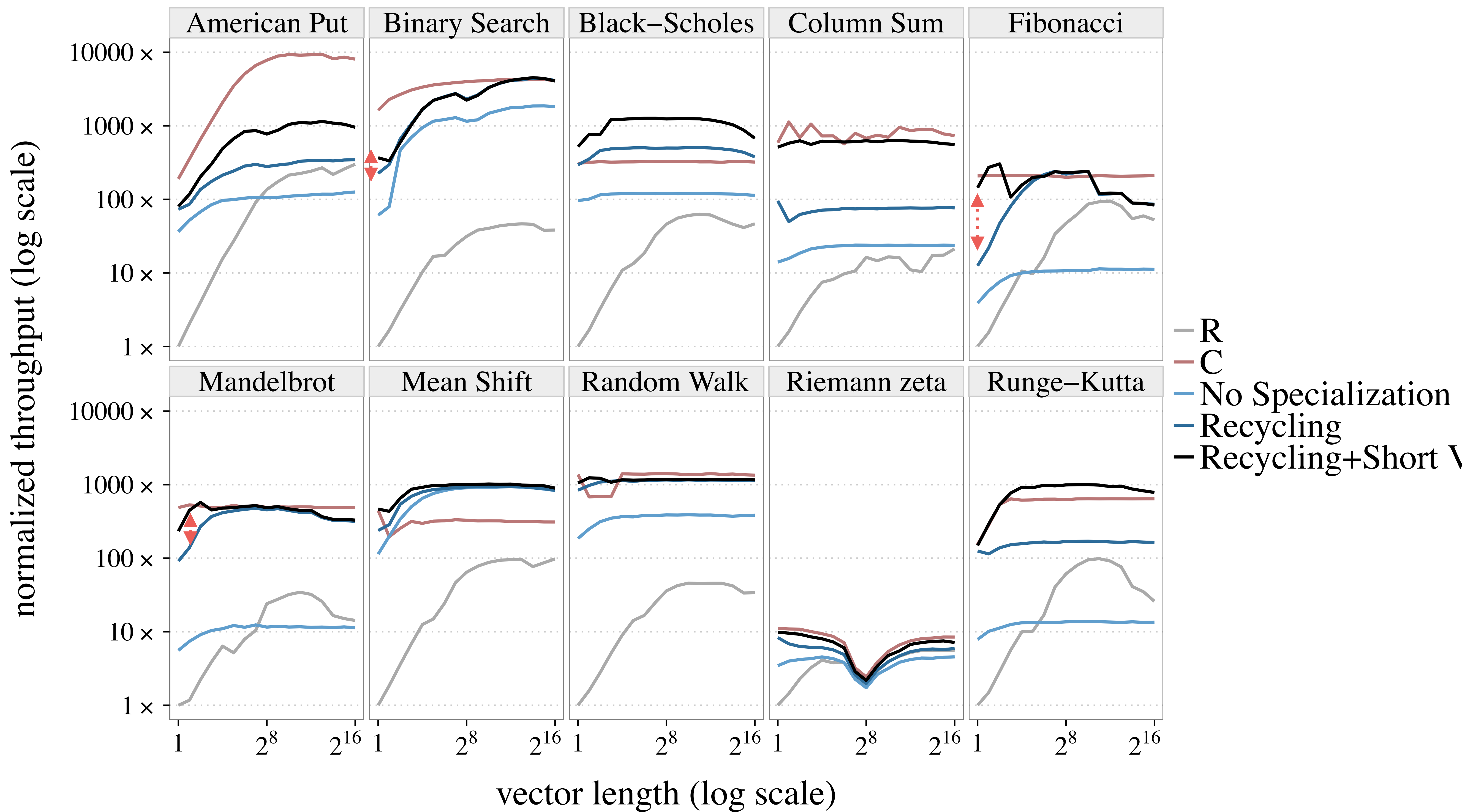

American Put
Binary Search
Black−Scholes
Column Sum
Fibonacci
Mandelbrot
Mean Shift
Random Walk
Riemann zeta
Runge−Kutta
10000 ×
1000 ×
100 ×
10 ×
1 ×
1
2^8
2^{16}
normalized throughput (log scale)
vector length (log scale)
R
C
No Specialization

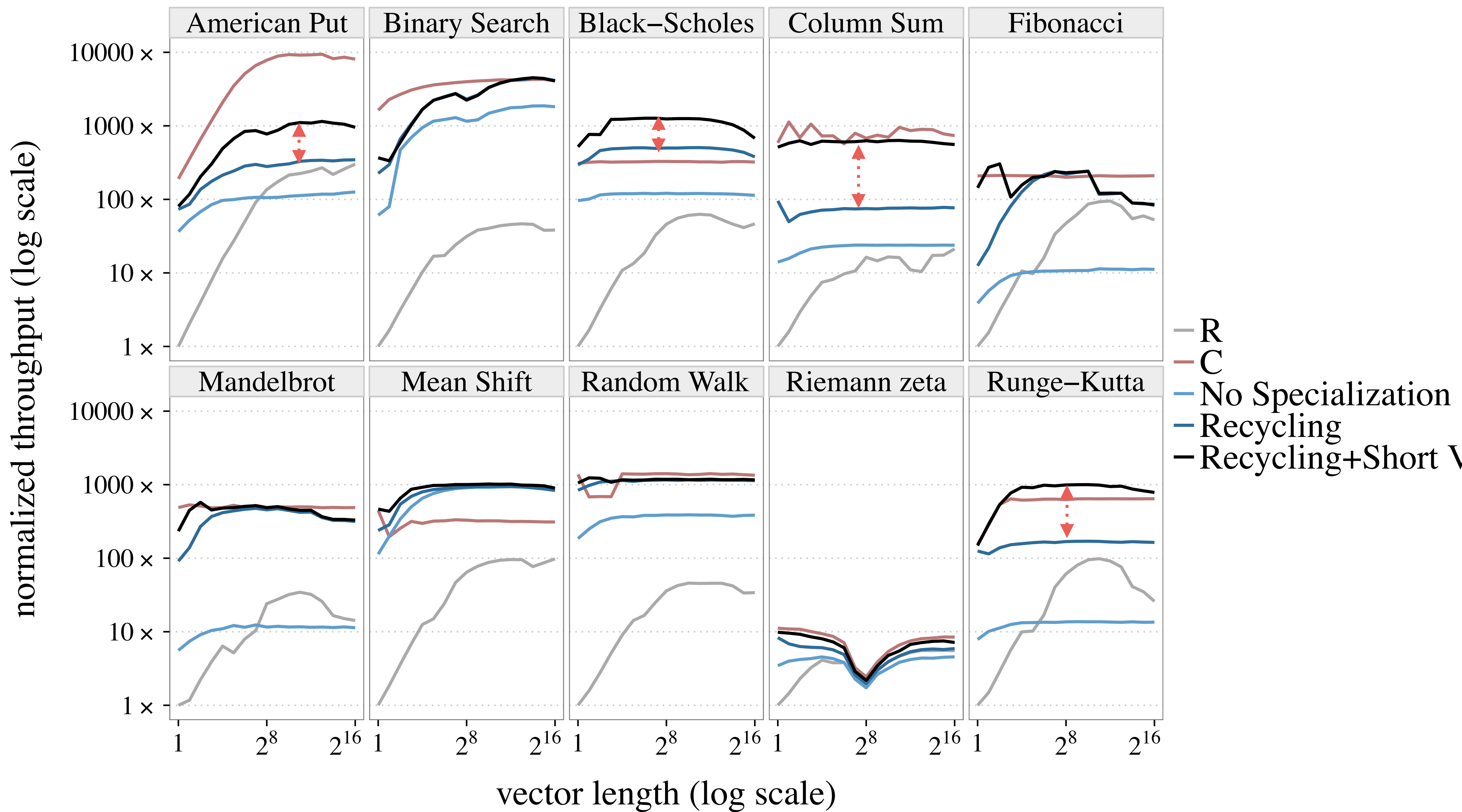

American Put
Binary Search
Black–Scholes
Column Sum
Fibonacci
Mandelbrot
Mean Shift
Random Walk
Riemann zeta
Runge–Kutta
10000 ×
1000 ×
100 ×
10 ×
1 ×
normalized throughput (log scale)
1
2^8
2^16
vector length (log scale)
R
C
No Specialization


American Put
Binary Search
Black−Scholes
Column Sum
Fibonacci
Mandelbrot
Mean Shift
Random Walk
Riemann zeta
Runge−Kutta
10000 ×
1000 ×
100 ×
10 ×
1 ×
normalized throughput (log scale)
1
2^8
2^16
vector length (log scale)
R
C
No Specialization
Recycling


American Put
Binary Search
Black–Scholes
Column Sum
Fibonacci
Mandelbrot
Mean Shift
Random Walk
Riemann zeta
Runge–Kutta
10000 ×
1000 ×
100 ×
10 ×
1 ×
1
2^8
2^16
normalized throughput (log scale)
vector length (log scale)
R
C
No Specialization
Recycling


American Put
Binary Search
Black–Scholes
Column Sum
Fibonacci
Mandelbrot
Mean Shift
Random Walk
Riemann zeta
Runge–Kutta
10000 ×
1000 ×
100 ×
10 ×
1 ×
1
2^8
2^16
normalized throughput (log scale)
vector length (log scale)
R
C
No Specialization
Recycling


American Put
Binary Search
Black–Scholes
Column Sum
Fibonacci
Mandelbrot
Mean Shift
Random Walk
Riemann zeta
Runge–Kutta
10000 ×
1000 ×
100 ×
10 ×
1 ×
normalized throughput (log scale)
1
2^8
2^{16}
vector length (log scale)
R
C
No Specialization
Recycling
Recycling+Short V


American Put
Binary Search
Black−Scholes
Column Sum
Fibonacci
Mandelbrot
Mean Shift
Random Walk
Riemann zeta
Runge−Kutta
10000 ×
1000 ×
100 ×
10 ×
1 ×
1
2^8
2^{16}
normalized throughput (log scale)
vector length (log scale)
R
C
No Specialization
Recycling
Recycling+Short V

How far did we get?

How far did we get?

- More stable performance across a wide-range of vector sizes, *but not yet as good as hand-written C on some workloads*.
- Performance on-par with C for some workloads, *but not all*.
    - Faster when we can make better use of SSE
    - Slower when there is scalar control flow

Open issues

Incomplete story

- Instrumentation showed our heuristics will not increase compilation overhead “much”
- Evaluation showed specialization with our heuristics increases performance across a wide range of vector lengths
- Missing: Real-world workloads running in Riposte to demonstrate that our approach works in the wild.

Long vs. short

- Unify long/short vector strategies in a single JIT?
  - Deferred vs hot loop execution?
  - Medium length vectors?
  - What can we learn from nested parallel languages?

LLVM

Stage	**Time (s)**	**Percent**
Early optimizations	0.003	3.2%
Length specialization	≤ 0.001	$\sim 0.5\%$
Vector optimizations	≤ 0.001	$\sim 0.5\%$
Generating LLVM instructions	0.002	2.2%
LLVM optimization passes	0.012	13.0%
LLVM code emission	0.074	80.4%

Table 1. Compilation time for BLACK-SCHOLES.

Current State of Riposte

Towards Completeness

- Much harder than I originally thought…and I was originally pessimistic
- 700 Primitive & Internal functions
  - many not documented at all…what does `.addCondHands` do?
  - Riposte implements most of these in R (including S3 dispatch)
  - Riposte has ~80 primitive functions, most much lower level than R's
- FFI
  - R header files (Rinternals.h, argh!) expose way too much of the internal implementation details

Vector FFIs?

`.Map(ff_name, ...)`

Runtime handles recycling arguments and calls ff_name to get each result.

`.Reduce(ff_name, base_case, ...)`

Runtime handles iteration

Vector FFIs?

- Runtime can do vector optimizations such as fusion
- Runtime can parallelize FFI execution
- Many built-in functions could be moved to libraries (e.g. transcendental functions)

Thanks