Hewlett Packard
Enterprise

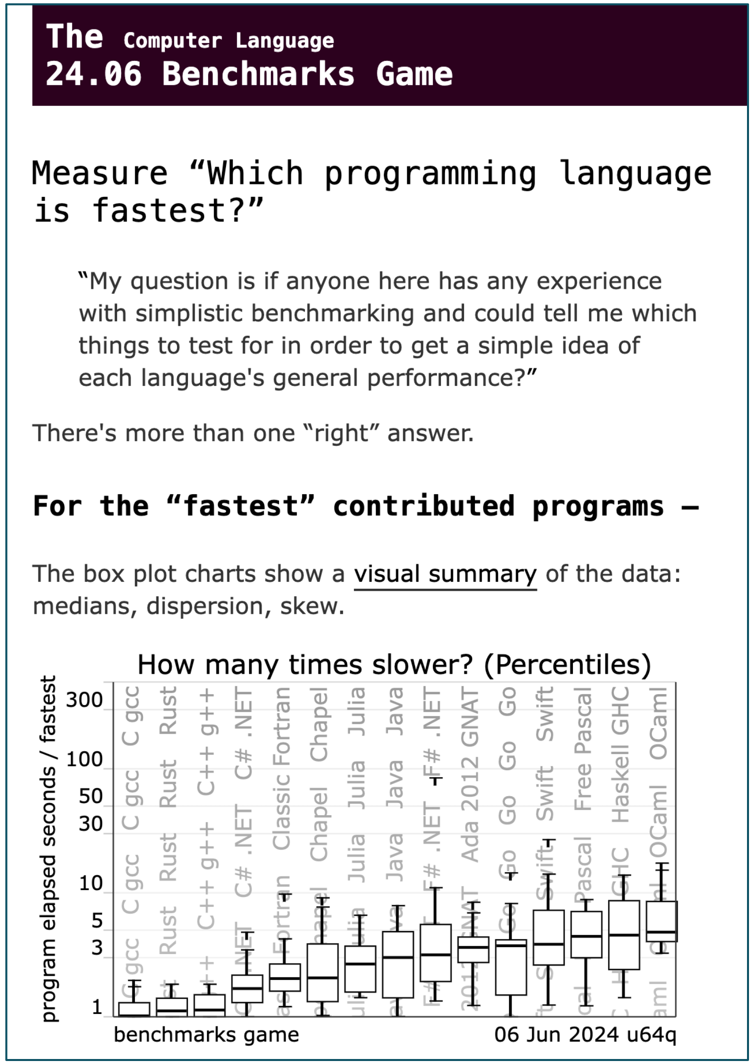
The Computer Language Benchmarks Game and Chapel 2.0

Brad Chamberlain

ChapelCon'24, June 7, 2024

What is the Computer Language Benchmarks Game (CLBG)?

- A website comparing a few dozen languages using 10 benchmarks
  - Benchmarks exercise useful things like:
    - floating point performance
    - IO
    - vectorization
    - bigints
    - …
  - Supports comparisons in terms of:
    - **wallclock time**
    - memory usage
    - **code compactness**
    - CPU time
    - CPU load
    - **browsing the source code** (encouraged, but obvs. requires effort)
  - Accepts new code submissions *of the same algorithm*


The Computer Language
24.06 Benchmarks Game

Measure "Which programming language is fastest?"

"My question is if anyone here has any experience with simplistic benchmarking and could tell me which things to test for in order to get a simple idea of each language's general performance?"

There's more than one "right" answer.

For the "fastest" contributed programs –

The box plot charts show a visual summary of the data: medians, dispersion, skew.

https://benchmarksgame-team.pages.debian.net/benchmarksgame/

Chapel's approach to the CLBG

- **Our Goal:** Submit versions that are fast but clear
  - Strive for versions that would be great to learn from

- Use results to understand where Chapel falls short
  - in terms of performance
  - in terms of expressiveness / capabilities

The Computer Language
24.06 Benchmarks Game

all Chapel programs & measurements

File system caches and swap are cleared before measurements are made for each program — so each program has a similiar initial context. That makes the first measurements (the smallest N workload) different from later measurements.

chpl version 2.0.0
built with LLVM version 17.0.2
Copyright 2020-2024
Hewlett Packard Enterprise Development LP
Copyright 2004-2019 Cray Inc.

source	secs	N	mem	gz	cpu secs	cpu load
binary-trees #3	0.34	7	19,568	494	0.02	0% 0% 5% 2%
binary-trees #3	0.06	14	19,568	494	0.15	100% 85% 66% 57%
binary-trees #3	8.71	21	367,232	494	26.20	99% 74% 56% 71%

source	secs	N	mem	gz	cpu secs	cpu load
fannkuch-redux #2	0.32	10	19,596	737	0.23	15% 21% 18% 24%
fannkuch-redux #2	0.64	11	19,596	737	2.53	100% 98% 98% 100%
fannkuch-redux #2	8.40	12	19,596	737	33.50	100% 100% 99% 99%

source	secs	N	mem	gz	cpu secs	cpu load

https://benchmarksgame-team.pages.debian.net/benchmarksgame/measurements/chapel.html

Reading a Benchmark's Results

- Each benchmark has its own results page:
  - Here, we're looking at spectral-norm
  - Click on "description" to learn about it
- Starts with a few simple/clear versions:
  - (good ones to learn the algorithm from)
- Then, the pack of main contenders:

The Computer Language
24.06 Benchmarks Game

spectral-norm

description

First a few simple programs.
Then optimisations, multicore parallelism, [pdf] vector parallelism.
Last hand-written vector instructions and "unsafe" programs.

source	secs	mem	gz
Julia #2	1.36	258,688	377
Go #4	1.43	20,340	555
Chapel	1.46	19,688	322

×	source	secs	mem	gz	cpu secs	cpu load
1.0	**Rust** #5	**0.72**	19,748	1062	2.85	100% 100% 100% 100%
1.0	Rust #7	0.72	19,748	938	2.85	100% 100% 100% 100%
1.0	Classic **Fortran** #3	**0.72**	19,652	644	2.85	100% 100% 98% 100%
1.0	Rust #4	0.72	19,812	823	2.85	98% 98% 100% 100%
1.0	**Chapel** #2	**0.73**	19,688	348	2.88	100% 98% 100% 100%
1.7	**Julia** #4	**1.19**	251,184	435	3.64	75% 99% 64% 67%
1.9	Julia #2	1.36	258,688	377	4.07	89% 64% 75% 71%
2.0	**Swift** #3	**1.43**	20,084	607	5.69	100% 99% 100% 99%
2.0	**Go** #4	**1.43**	20,340	555	5.68	99% 99% 99% 99%
2.0	**C** gcc #3	**1.43**	19,708	470	5.70	100% 100% 100% 100%
2.0	**Lisp** SBCL #8	**1.44**	19,688	799	5.64	98% 99% 98% 98%
2.0	Free **Pascal** #2	**1.44**	19,688	548	5.71	99% 99% 98% 98%

spectral-norm description

program measurements

Background

MathWorld: "Hundred-Dollar, Hundred-Digit Challenge Problems", Challenge #3.

Thanks to Sebastien Loisel for suggesting this task.

How to implement

We ask that contributed programs not only give the correct result, but also **use the same algorithm** to calculate that result.

Each program should:

- calculate the spectral norm of an infinite matrix A, with entries $a_{11}=1$, $a_{12}=1/2$, $a_{21}=1/3$, $a_{13}=1/4$, $a_{22}=1/5$, $a_{31}=1/6$, etc
- implement 4 separate functions / procedures / methods like the Java program

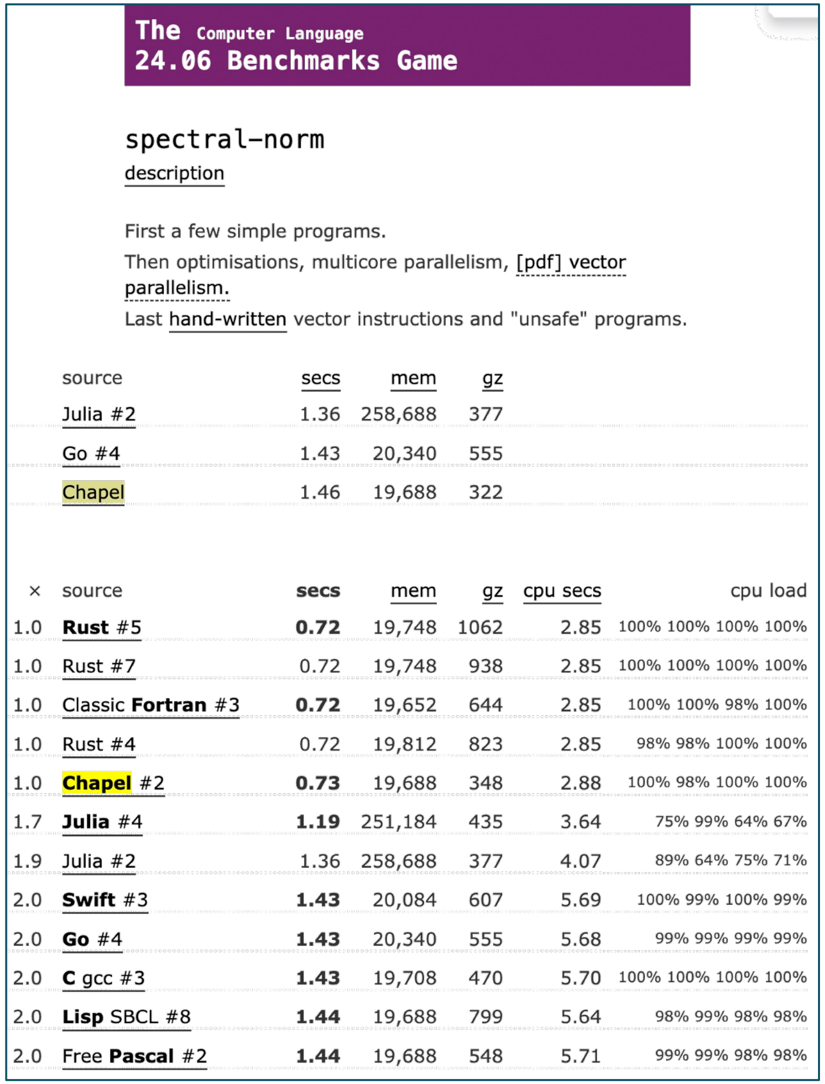
diff program output N = 100 with this output file to check your program output has the correct format, before you contribute your program.

Use a larger command line argument (5500) to check program performance.

https://benchmarksgame-team.pages.debian.net/benchmarksgame/performance/spectralnorm.html

Reading a Benchmark's Results

- By default, entries are sorted by 'secs'
  - (wall-clock time)
- This Chapel #2 entry took 0.73 seconds
  - and essentially runs in 1.0x of the baseline
    - (the Rust #5 version at the top)
- Click on a heading to change the sort…
  - e.g., 'gz' (code compactness)


The Computer Language
24.06 Benchmarks Game

spectral-norm

description

First a few simple programs.
Then optimisations, multicore parallelism, [pdf] vector parallelism.
Last hand-written vector instructions and "unsafe" programs.

source	secs	mem	gz
Julia #2	1.36	258,688	377
Go #4	1.43	20,340	555
Chapel	1.46	19,688	322

×	source	secs	mem	gz	cpu secs	cpu load
1.0	**Rust** #5	**0.72**	19,748	1062	2.85	100% 100% 100% 100%
1.0	Rust #7	0.72	19,748	938	2.85	100% 100% 100% 100%
1.0	Classic **Fortran** #3	**0.72**	19,652	644	2.85	100% 100% 98% 100%
1.0	Rust #4	0.72	19,812	823	2.85	98% 98% 100% 100%
1.0	**Chapel** #2	**0.73**	19,688	348	2.88	100% 98% 100% 100%
1.7	**Julia** #4	**1.19**	251,184	435	3.64	75% 99% 64% 67%
1.9	Julia #2	1.36	258,688	377	4.07	89% 64% 75% 71%
2.0	**Swift** #3	**1.43**	20,084	607	5.69	100% 99% 100% 99%
2.0	**Go** #4	**1.43**	20,340	555	5.68	99% 99% 99% 99%
2.0	**C** gcc #3	**1.43**	19,708	470	5.70	100% 100% 100% 100%
2.0	**Lisp** SBCL #8	**1.44**	19,688	799	5.64	98% 99% 98% 98%
2.0	**Free Pascal** #2	**1.44**	19,688	548	5.71	99% 99% 98% 98%

https://benchmarksgame-team.pages.debian.net/benchmarksgame/performance/spectralnorm.html

Reading a Benchmark's Results

- Sorting by code compactness…

The Computer Language
24.06 Benchmarks Game

spectral-norm

description

First a few simple programs.

Then optimisations, multicore parallelism, [pdf] vector parallelism.

Last hand-written vector instructions and "unsafe" programs.

source	secs	mem	gz
Chapel	1.46	19,688	322
Julia #2	1.36	258,688	377
Go #4	1.43	20,340	555

×	source	secs	mem	gz	cpu secs	cpu load
1.0	Matz's **Ruby**	26 min	11,056	**292**	26 min	54% 1% 32% 14%
1.0	**Ruby** yjit	128.41	22,016	**299**	128.41	0% 0% 100% 0%
1.1	**Chapel**	1.46	19,688	**322**	5.78	100% 99% 99% 99%
1.1	Matz's Ruby #4	29 min	11,056	326	29 min	34% 13% 26% 32%
1.1	**Node.js**	5.38	51,676	**326**	5.39	0% 0% 100% 0%
1.1	Ruby yjit #4	129.81	22,912	333	129.81	0% 0% 100% 0%
1.1	**Python 3** #6	5 min	19,660	**334**	5 min	0% 0% 100% 0%
1.1	**Lua**	78.68	19,652	**335**	78.68	35% 64% 0% 0%
1.2	**Perl**	104.08	19,652	**340**	104.08	0% 100% 0% 0%
1.2	Perl #5	97.67	19,828	346	97.66	0% 0% 0% 100%
1.2	Chapel #2	0.73	19,688	348	2.88	100% 98% 100% 100%
1.2	Perl #2	8 min	19,652	350	8 min	0% 100% 0% 0%

https://benchmarksgame-team.pages.debian.net/benchmarksgame/performance/spectralnorm.html

- We see another Chapel version that's 1.1x as compact as the baseline Ruby version

- Our Chapel #2 entry is 1.2x as compact
  - Demonstrating a speed::code size tension

Reading a Benchmark's Results

- Sorting by wall-clock time again…
  - Scrolling down, at the end…

…we find hand-written… / "unsafe" versions

- I refer to these as "heroic" for brevity
- Note these can outperform the baseline…

	Matz's Ruby #4	29 min	11,056	326	29 min	34% 13% 26% 32%
	C gcc #8	Make Error				
	F# .NET #2	Timed Out				

hand-written vector instructions | "unsafe"

×	source	**secs**	mem	gz	cpu secs	cpu load
0.5	C gcc #6	0.39	19,724	1203	1.54	100% 100% 100% 100%
1.0	C++ g++ #6	0.72	19,884	1050	2.85	100% 98% 100% 98%
1.0	Rust #6	0.72	19,780	1132	2.85	98% 100% 100% 100%
1.0	C gcc #5	0.72	19,708	576	2.86	100% 100% 100% 100%
1.0	C gcc #4	0.72	19,724	1145	2.85	100% 100% 98% 98%
1.0	C gcc #7	0.72	19,724	906	2.85	100% 100% 98% 98%
1.0	Ada 2012 GNAT #4	0.74	19,784	2777	2.86	97% 97% 97% 97%
1.1	Rust #2	0.78	19,748	1117	3.04	98% 98% 98% 98%
1.1	Rust	0.79	19,748	1262	3.02	98% 100% 98% 97%
1.3	Rust #3	0.92	19,748	1060	3.56	98% 98% 100% 98%
1.3	C# .NET #5	0.93	36,476	776	3.41	96% 92% 90% 92%
1.9	C++ g++ #5	1.33	19,788	1050	5.27	100% 100% 100% 99%
5.5	Racket #3	3.91	76,412	639	14.84	93% 94% 99% 93%
21	Racket #2	15.10	75,252	539	15.10	0% 100% 0% 0%
31	Haskell GHC #2	22.30	19,688	410	22.48	0% 69% 31% 0%

by secs by mem by gz by cpu secs

How programs are measured

https://benchmarksgame-team.pages.debian.net/benchmarksgame/performance/spectralnorm.html

Reading a Benchmark's Results

- Scolling back up…
  - Let's find the other Chapel version's timings

source	secs	mem	gz
Julia #2	1.36	258,688	377
Go #4	1.43	20,340	555
Chapel	1.46	19,688	322

×	source	secs	mem	gz	cpu secs	cpu load
1.0	**Rust** #5	**0.72**	19,748	1062	2.85	100% 100% 100% 100%
1.0	Rust #7	0.72	19,748	938	2.85	100% 100% 100% 100%
1.0	Classic **Fortran** #3	**0.72**	19,652	644	2.85	100% 100% 98% 100%
1.0	Rust #4	0.72	19,812	823	2.85	98% 98% 100% 100%
1.0	**Chapel** #2	**0.73**	19,688	348	2.88	100% 98% 100% 100%
1.7	**Julia** #4	**1.19**	251,184	435	3.64	75% 99% 64% 67%
1.9	Julia #2	1.36	258,688	377	4.07	89% 64% 75% 71%
2.0	**Swift** #3	**1.43**	20,084	607	5.69	100% 99% 100% 99%
2.0	**Go** #4	**1.43**	20,340	555	5.68	99% 99% 99% 99%
2.0	**C** gcc #3	**1.43**	19,708	470	5.70	100% 100% 100% 100%
2.0	**Lisp** SBCL #8	**1.44**	19,688	799	5.64	98% 99% 98% 98%
2.0	Free **Pascal** #2	**1.44**	19,688	548	5.71	99% 99% 98% 98%
2.0	Free Pascal #3	1.45	19,688	656	5.71	98% 99% 98% 99%
2.0	Lisp SBCL #2	1.45	19,688	920	5.64	98% 98% 99% 99%
2.0	**Dart** #6	**1.45**	19,972	1202	5.70	98% 98% 98% 98%
2.0	Lisp SBCL #3	1.46	19,688	893	5.63	98% 99% 98% 97%
2.0	Chapel	1.46	19,688	322	5.78	100% 99% 99% 99%
2.0	Lisp SBCL #7	1.46	19,688	769	5.65	97% 98% 99% 97%
2.1	**Ada** 2012 GNAT #3	**1.47**	19,784	1725	5.73	98% 97% 97% 98%
2.1	**Haskell** GHC #4	**1.48**	19,688	994	5.72	96% 98% 96% 97%
2.1	Go #2	1.50	20,148	674	5.69	94% 94% 96% 94%

- Here it is…
  - …2.0 slower than the Rust baseline:

So we have…

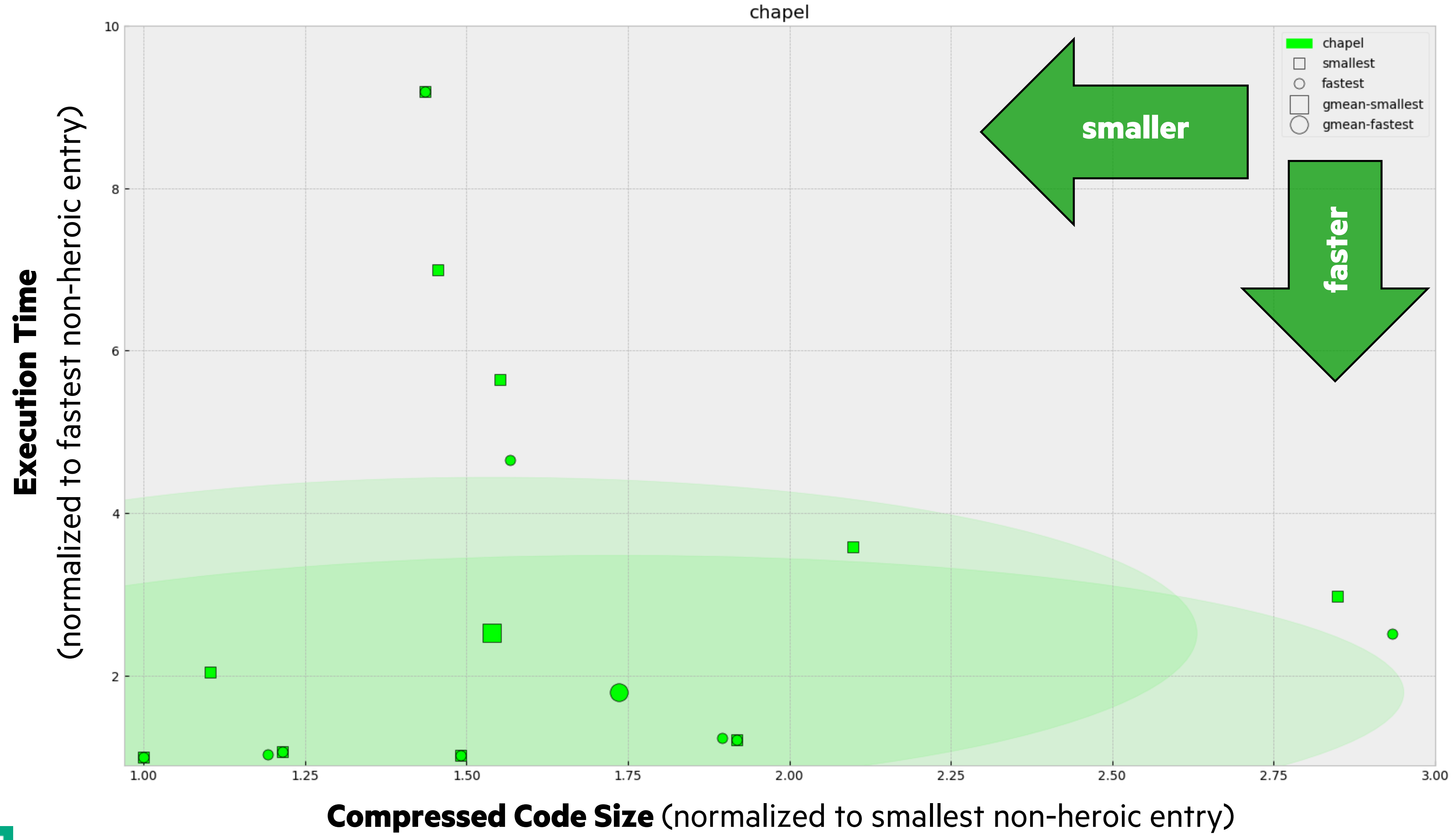
…Chapel: 2.0x slower, 1.1x less compact

…Chapel #2: 1.0x slower, 1.2x less compact

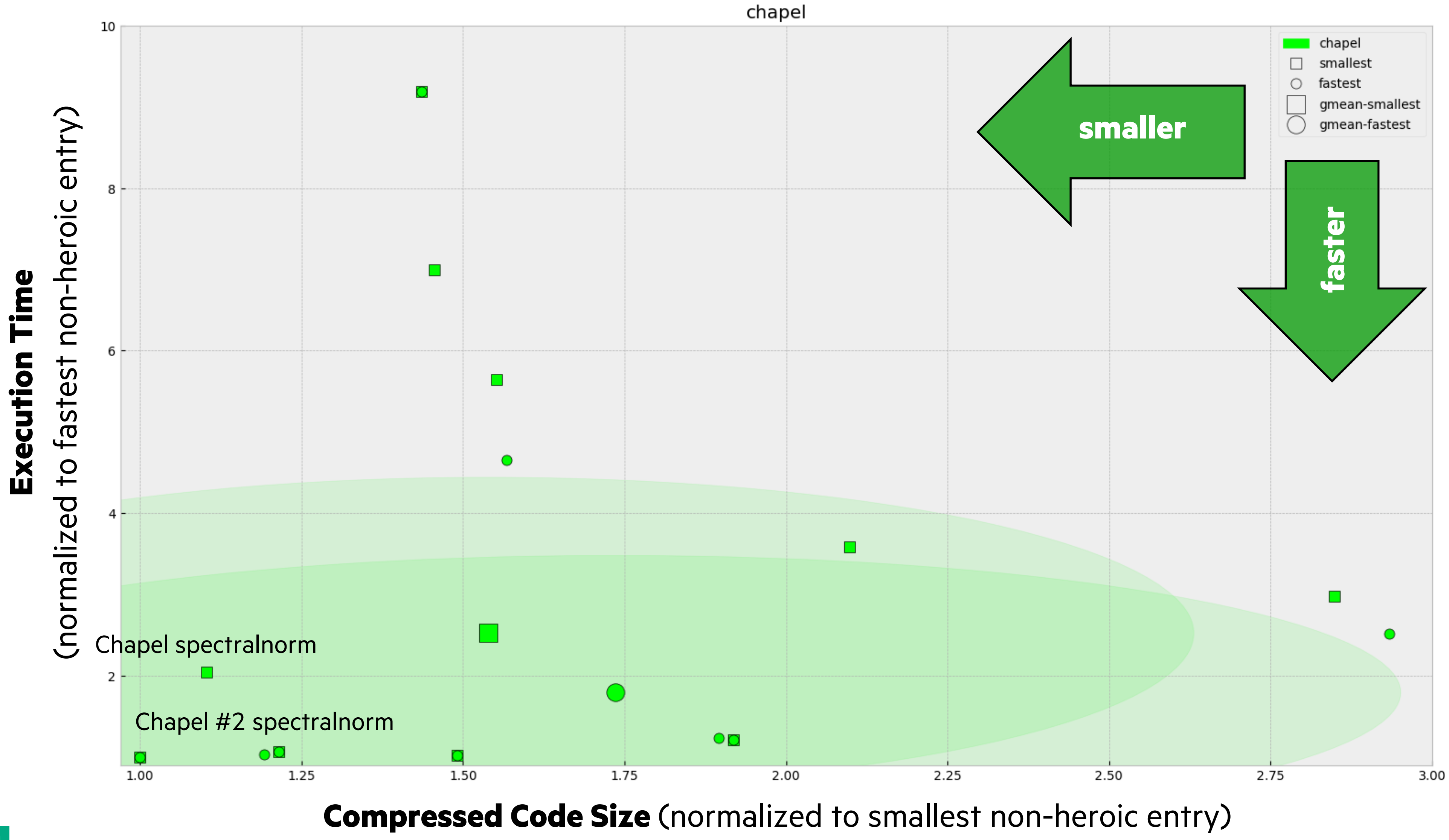
Let's plot this tension!

https://benchmarksgame-team.pages.debian.net/benchmarksgame/performance/spectralnorm.html

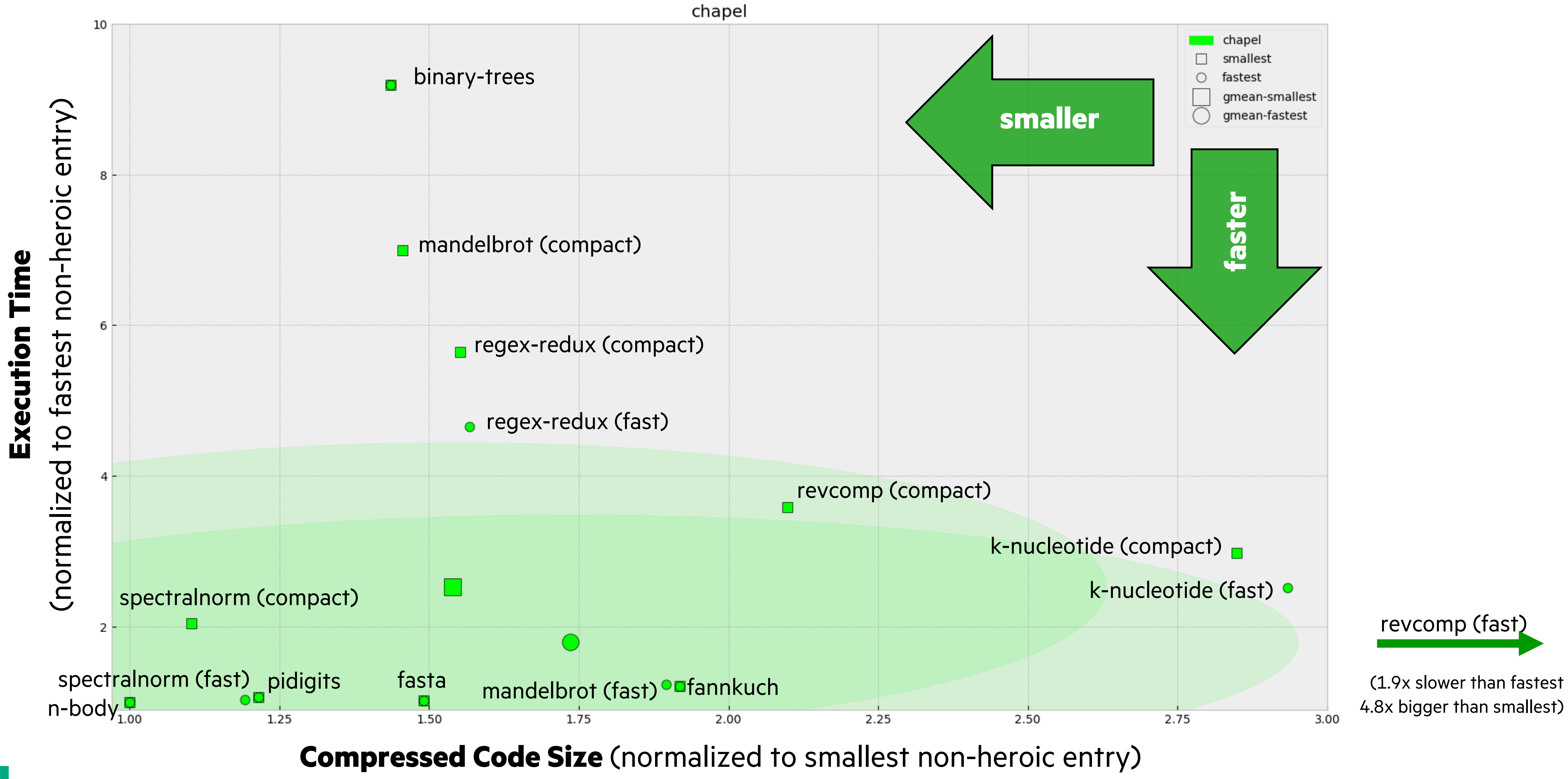
CLBG: Scatter Plot of Chapel's fastest/most-compact benchmarks (Apr 5, 2024)



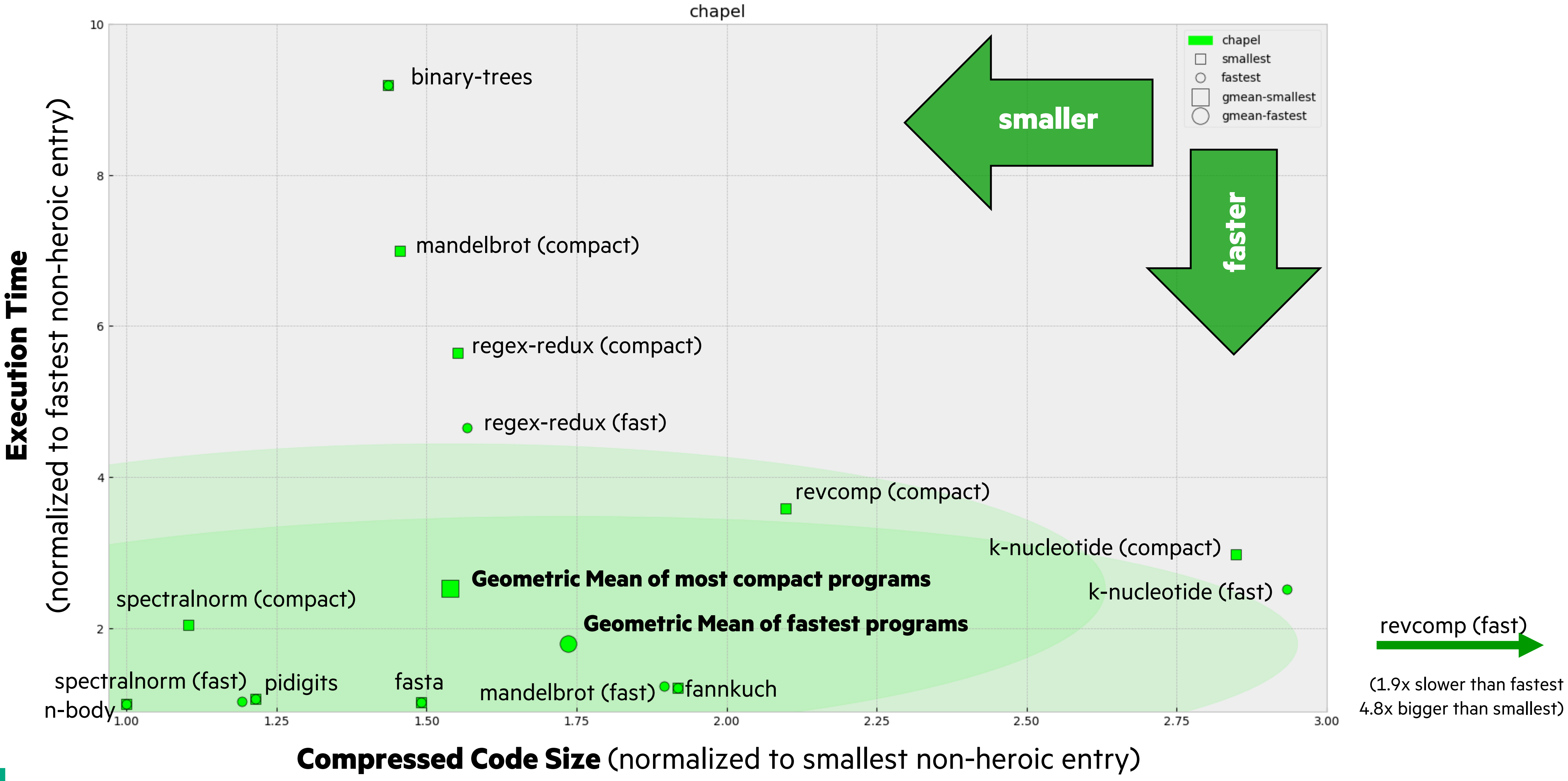
CLBG: Chapel's fastest/most-compact versions of spectral norm (Apr 5, 2024)



CLBG: Chapel's fastest/most-compact versions of all benchmarks (Apr 5, 2024)

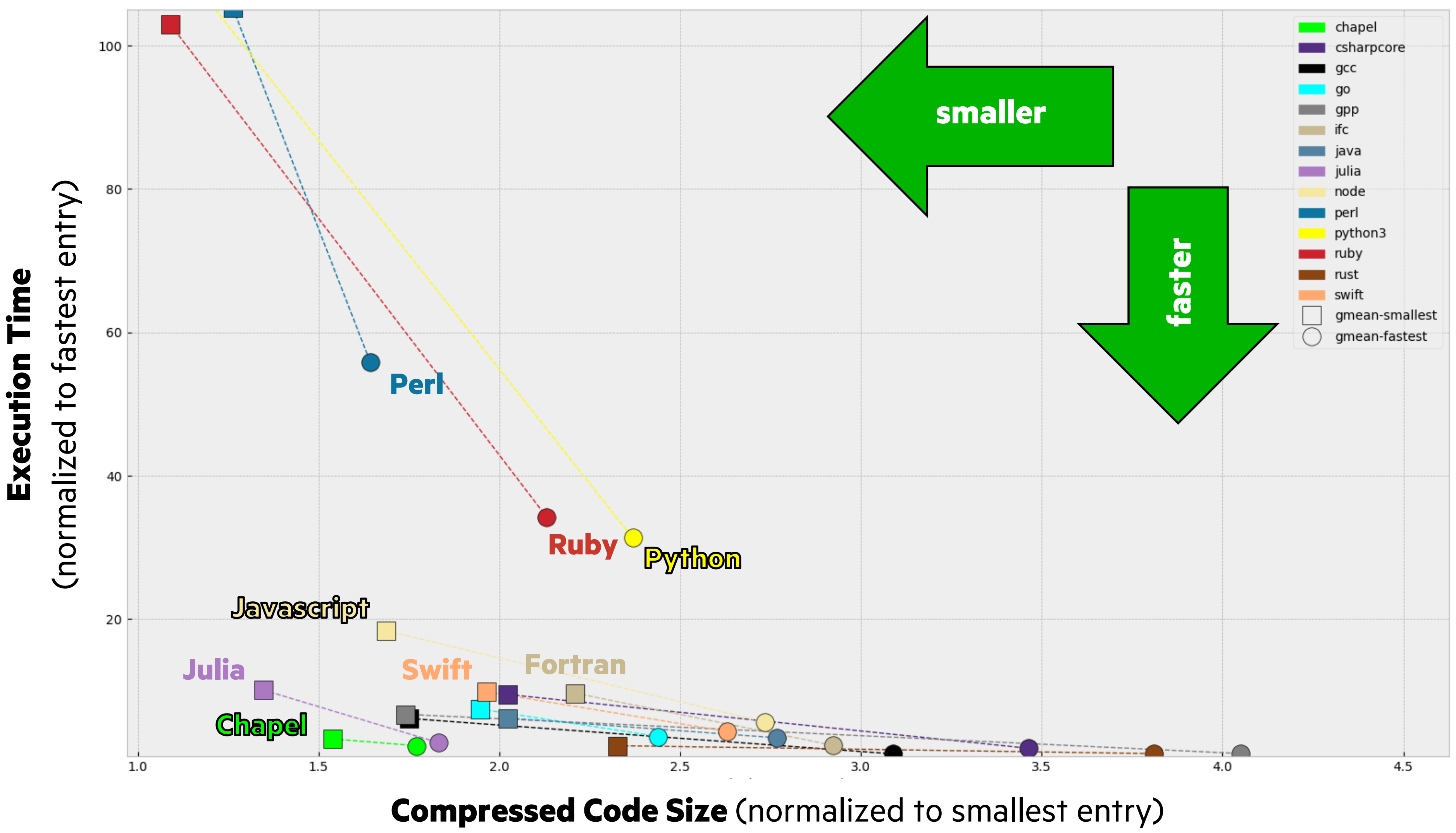


CLBG: Geometric Means of Chapel's fastest/most-compact versions (Apr 5, 2024)



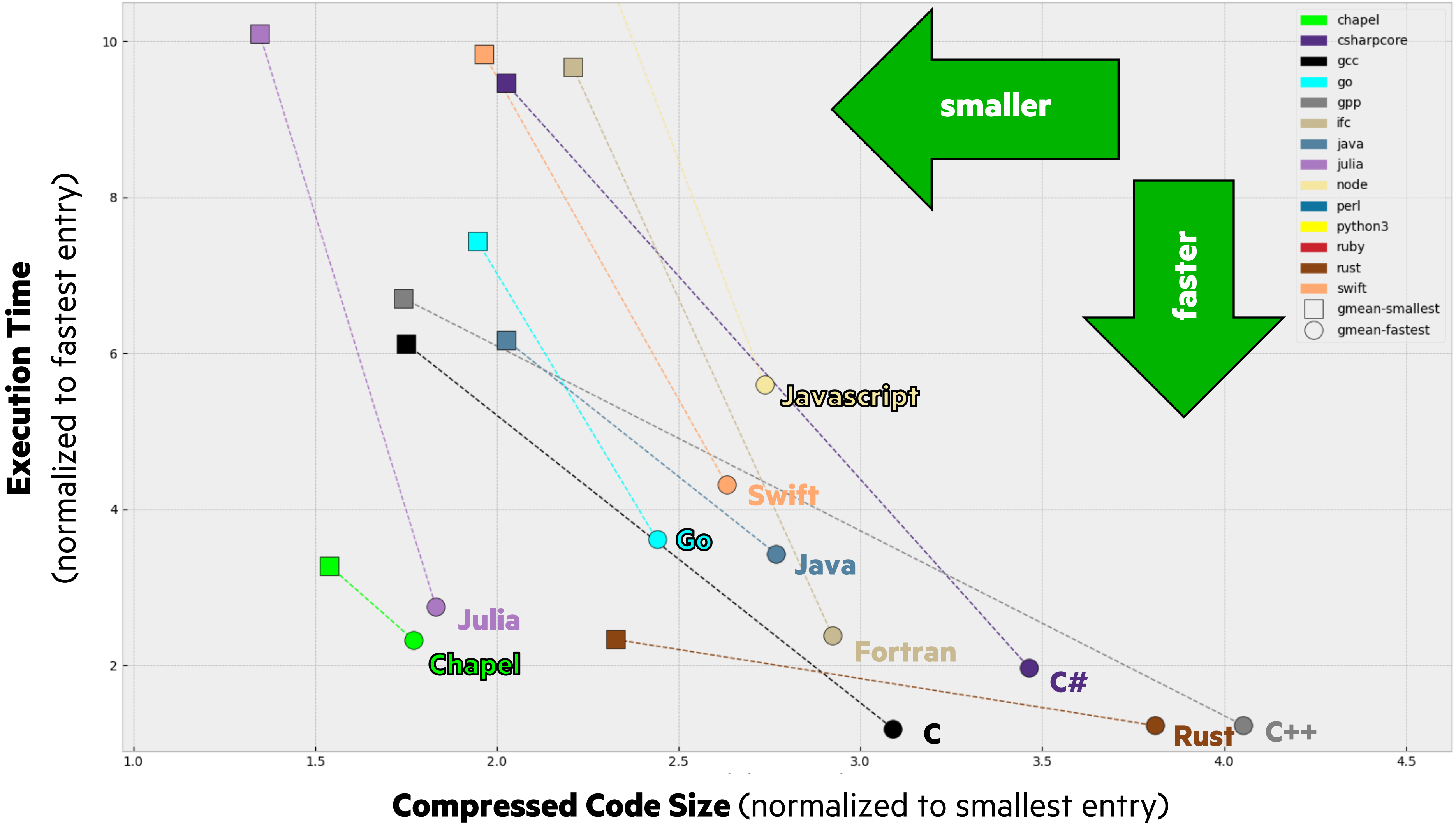
We can then use these geometric means to summarize each language compactly…

CLBG Summary, Apr 12, 2024 (selected languages, w/ heroic versions)



Note: Regrettably, the version of this chart presented at ChapelCon '24 included incorrect summary results for C, C#, Go, Java, Perl, Python, and Ruby due to a bug in our scripts; this is the corrected version

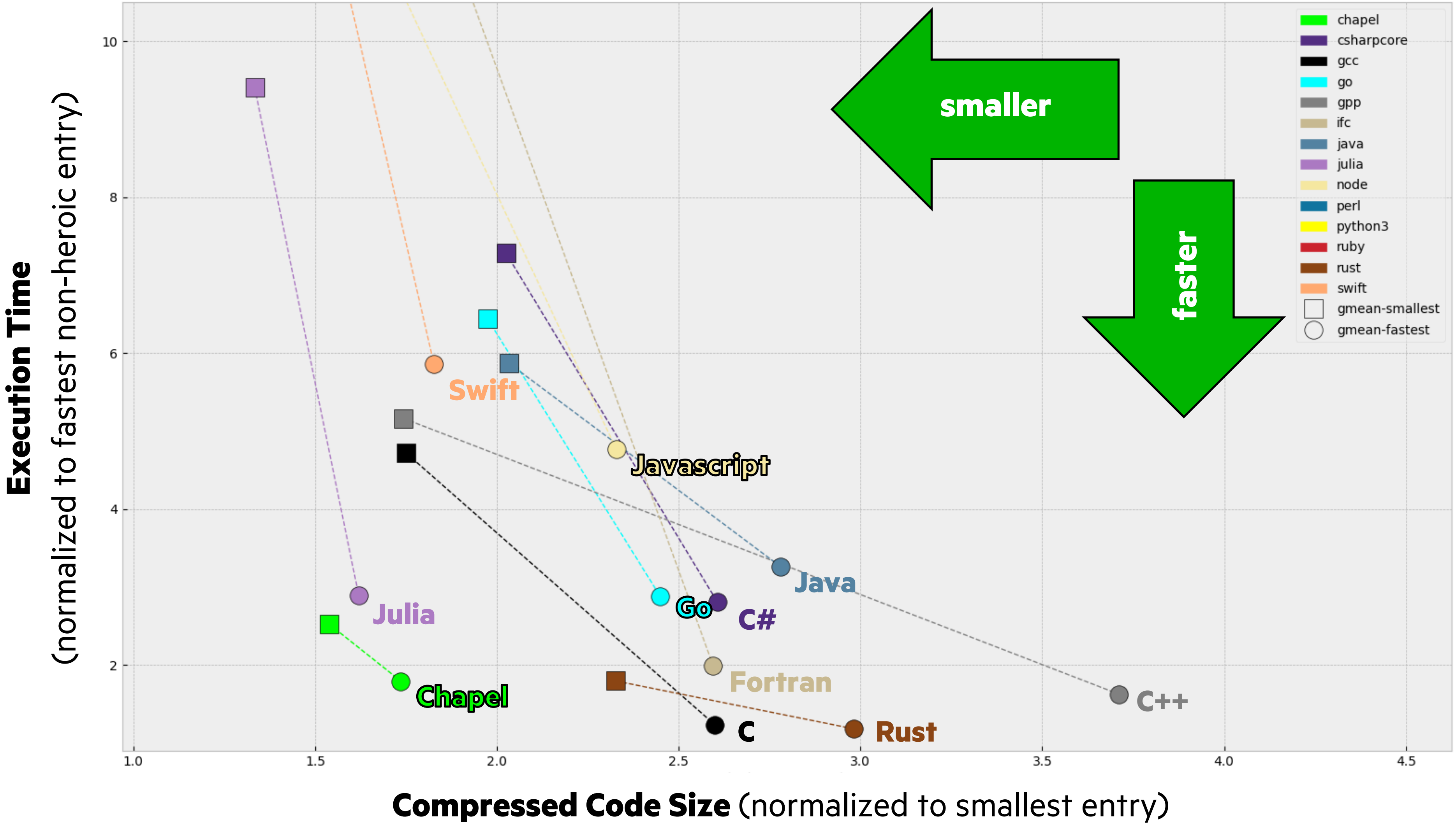
CLBG Summary, Apr 12, 2024 (selected languages, w/ heroic versions, zoomed-in)



Note: Regrettably, the version of this chart presented at ChapelCon '24 included incorrect summary results for C, C#, Go, and Java due to a bug in our scripts; this is the corrected version

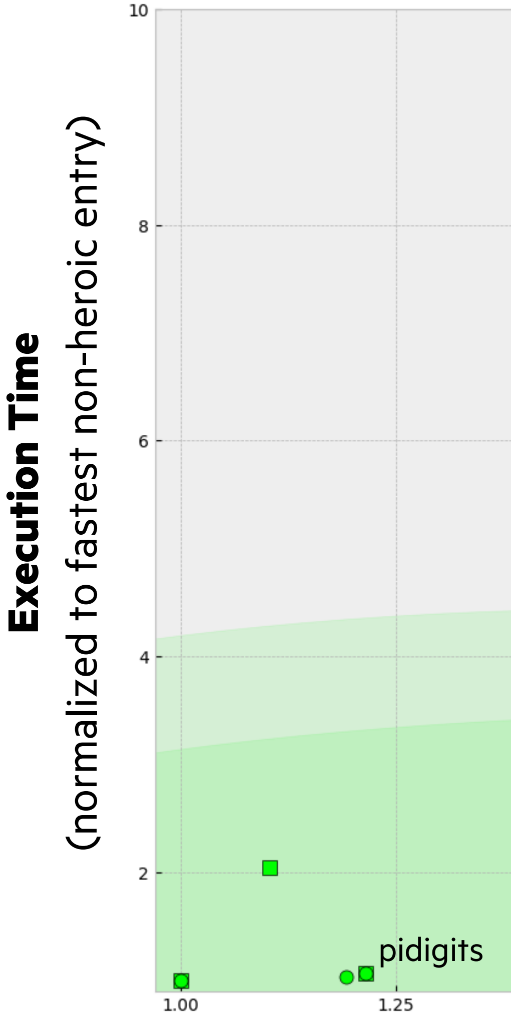
Those graphs included the heroic versions; removing those…

CLBG Summary, Apr 12, 2024 (selected languages, no heroic versions, zoomed-in)



Note: Regrettably, the version of this chart presented at ChapelCon '24 included incorrect summary results for C, C#, Go, and Java due to a bug in our scripts; this is the corrected version

CLBG: Often, a single version is both Chapel's fastest and most compact



The Computer Language
24.06 Benchmarks Game

pidigits

description

Arbitrary precision arithmetic might be provided by wrapping a third-party library written in some-other programming language. How would you know?

First a few simple programs.

Then optimisations, multicore parallelism, [pdf] vector parallelism.

Last hand-written vector instructions and "unsafe" programs and the more obvious foreign function interface programs.

source	secs	mem	gz
Haskell GHC #6	1.62	19,688	368
Lisp SBCL #3	3.49	616,192	499
Racket	10.41	77,952	459

×	source	secs	mem	gz	cpu secs	cpu load
1.0	**Rust** #4	**0.71**	19,720	804	0.71	1% 100% 0% 1%
1.1	**Chapel** #2	**0.76**	19,976	423	0.77	98% 5% 2% 1%
1.1	**C** gcc #2	**0.82**	19,704	422	0.82	100% 1% 1% 1%
1.2	C gcc	0.89	19,704	459	0.88	1% 100% 0% 1%
1.2	**C++** g++ #4	**0.89**	19,736	521	0.88	1% 1% 1% 100%
1.4	**PHP** #5	**1.03**	19,656	405	1.03	1% 1% 100% 0%
1.5	PHP #4	1.04	19,656	396	1.04	100% 0% 0% 1%
1.5	PHP #3	1.05	19,656	510	1.05	0% 100% 0% 0%
1.7	**Node.js** #4	**1.23**	56,100	487	1.26	0% 0% 2% 99%
1.9	**Go**	**1.34**	19,700	715	1.36	75% 1% 0% 27%

The Computer Language
24.06 Benchmarks Game

pidigits

description

Arbitrary precision arithmetic might be provided by wrapping a third-party library written in some-other programming language. How would you know?

First a few simple programs.

Then optimisations, multicore parallelism, [pdf] vector parallelism.

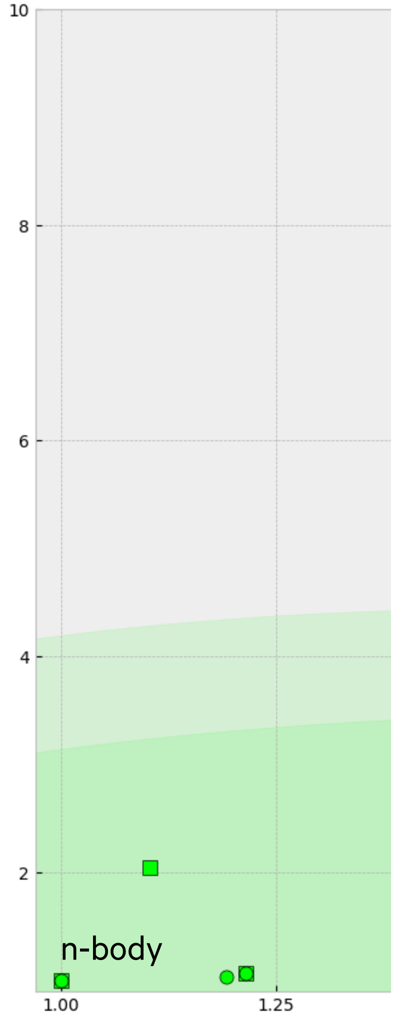
Last hand-written vector instructions and "unsafe" programs and the more obvious foreign function interface programs.

source	secs	mem	gz
Haskell GHC #6	1.62	19,688	368
Racket	10.41	77,952	459
Lisp SBCL #3	3.49	616,192	499

×	source	secs	mem	gz	cpu secs	cpu load
1.0	**Python 3** #4	4.61	19,652	**348**	4.61	0% 0% 99% 0%
1.0	**Haskell** GHC #4	1.83	19,688	**355**	1.89	66% 6% 2% 27%
1.1	Haskell GHC #6	1.62	19,688	368	1.67	2% 75% 22% 2%
1.1	Haskell GHC #3	2.21	19,688	387	2.28	36% 60% 1% 2%
1.1	**PHP** #4	1.04	19,656	**396**	1.04	100% 0% 0% 1%
1.2	PHP #5	1.03	19,656	405	1.03	1% 1% 100% 0%
1.2	**Node.js** #2	12.45	84,544	**405**	12.47	0% 1% 99% 0%
1.2	**C** gcc #2	0.82	19,704	**422**	0.82	100% 1% 1% 1%
1.2	**Chapel** #2	0.76	19,976	**423**	0.77	98% 5% 2% 1%
1.2	Node.js #3	12.53	84,420	431	12.55	0% 99% 0% 0%

Compressed Code Size (normalized to smallest non-heroic entry)

CLBG: As of Chapel 2.0, our #3 n-body is the baseline for both speed and size!



Execution Time (normalized to fastest non-heroic entry)

Compressed Code Size (normalized to smallest non-heroic entry)

The Computer Language 24.06 Benchmarks Game

n–body

description

First a few simple programs.

Then optimisations, multicore parallelism, [pdf] vector parallelism.

Last hand-written vector instructions and "unsafe" programs.

source	secs	mem	gz
Chapel #3	3.89	19,672	967
C clang	5.52	19,672	1179
Java	7.89	41,464	1437

×	source	secs	mem	gz	cpu secs	cpu load
1.0	**Chapel** #3	**3.89**	19,672	967	3.90	100% 0% 0% 0%
1.0	**Rust** #2	**3.94**	19,700	1809	3.94	0% 0% 99% 0%
1.1	**Julia** #8	**4.17**	275,408	1129	4.26	2% 96% 0% 3%
1.1	Classic **Fortran** #6	**4.22**	19,644	1530	4.22	0% 0% 100% 1%
1.1	Rust #6	4.32	19,716	1796	4.32	0% 100% 0% 0%
1.2	Rust #8	4.49	19,684	1774	4.49	100% 0% 1% 0%
1.2	Chapel #4	4.58	19,656	1027	4.59	100% 0% 0% 0%

The Computer Language 24.06 Benchmarks Game

n–body

description

First a few simple programs.

Then optimisations, multicore parallelism, [pdf] vector parallelism.

Last hand-written vector instructions and "unsafe" programs.

source	secs	mem	gz
Chapel #3	3.89	19,672	967
C clang	5.52	19,672	1179
Java	7.89	41,464	1437

×	source	secs	mem	gz	cpu secs	cpu load
1.0	**Chapel** #3	3.89	19,672	**967**	3.90	100% 0% 0% 0%
1.0	Chapel #2	5.64	19,672	977	5.65	100% 1% 0% 0%
1.1	Chapel #4	4.58	19,656	1027	4.59	100% 0% 0% 0%
1.1	**Julia** #2	23.88	303,148	**1084**	24.01	0% 100% 0% 0%
1.1	**PHP** #3	67.03	19,656	**1088**	67.03	0% 0% 0% 100%
1.2	Julia #8	4.17	275,408	1129	4.26	2% 96% 0% 3%
1.2	Matz's **Ruby** #2	44 min	11,096	**1137**	44 min	0% 100% 0% 0%

Benchmark updates required by Chapel 2.0

	fasta	knucl	mandelbrot	pidigits2	regexredux	revcomp	spectralnorm
explicit ‘ref’ for passing arrays	X					X	X
reader( )/writer( ) signature updates	X	X				X	
read/writeBinary( ) updates	X					X	
readline( ) -> readLine( ) changes		X					
zip(keys, vals) instead of map.items( )		X					
sorted( ) iterator deprecated		X					
need to declare record ‘hashable’		X					
divCeilPos module/naming change			X				
bigint operator signature changes				X			
read(string) -> readAll( )					X		
compile(regex) -> new regex( )					X		
sub( ) -> replace( ) on regex					X		
change to lo..<hi type inference						X	
stricter C pointer aliasing rules						#8 only	

Unstable Features the current Chapel entries still rely on

	binarytrees	fannkuch2	knucleotide	mandelbrot	pidigits4	revcomp8
‘serial’ statement						X
divCeilPos( )				X		
‘DynamicIters’ module	X	X		X		
‘Sort’ module			X			
‘GMP’ module					X	

Opportunities for Future Improvement

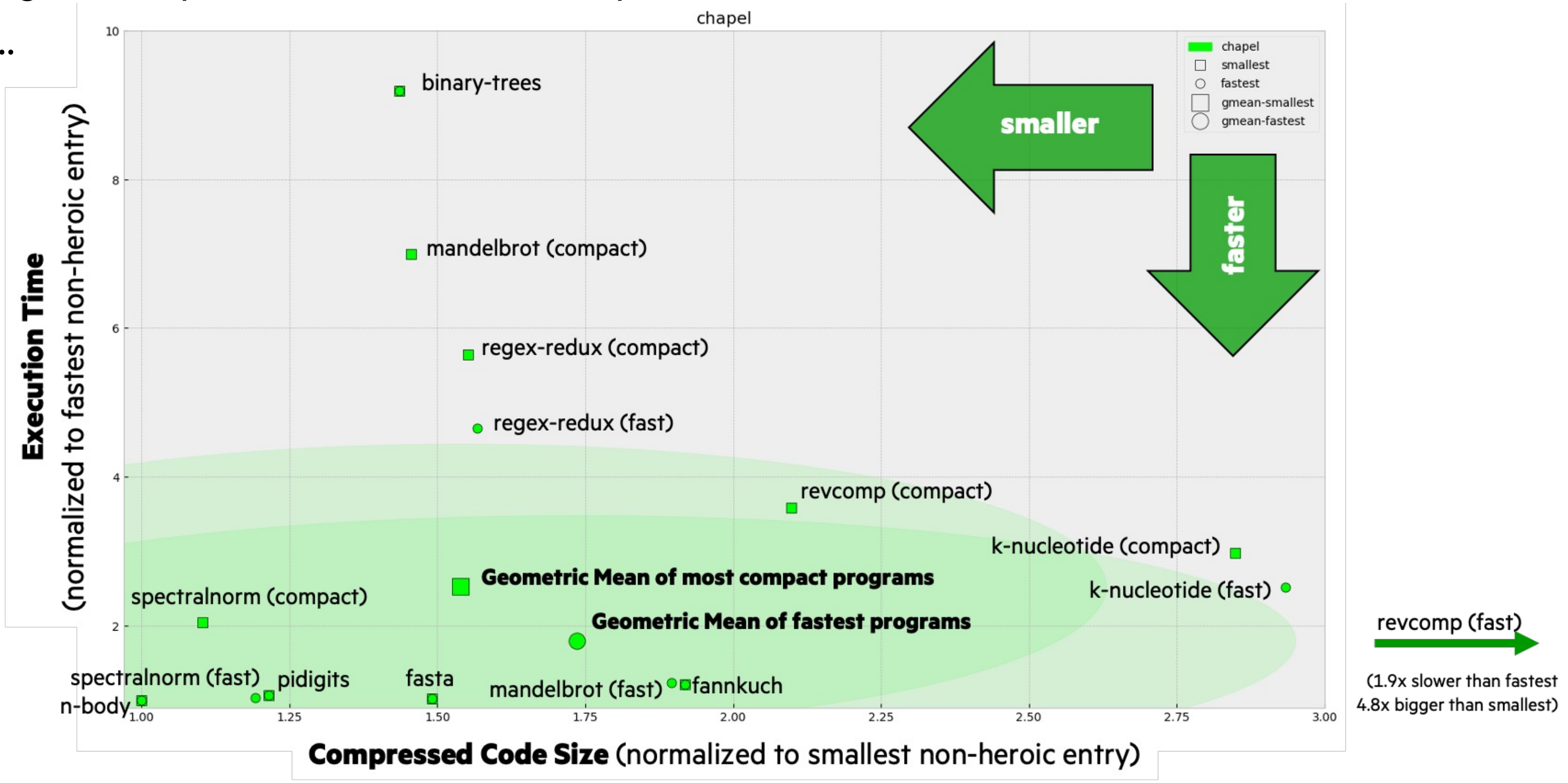
- **binary-trees:** Our worst outlier, due to lack of memory arenas / object pools / similar memory abstraction

- **regex-redux:**
  - Michael has already optimized some things in Chapel 2.1, so this should improve after it's released
  - Fastest entries use PCRE2, we use RE2…
    – should we switch?

- **revcomp, k-nucleotide:**
  - not doing great in either dimension…
  - I/O could be a place for improvement

- **nbody**, others…?:
  - written long ago
  - can be rewritten using modern Chapel

Caution: CLBG can be very addictive!



Thank you

https://chapel-lang.org
@ChapelLanguage