Massachusetts Institute of Technology

MIT CSAIL

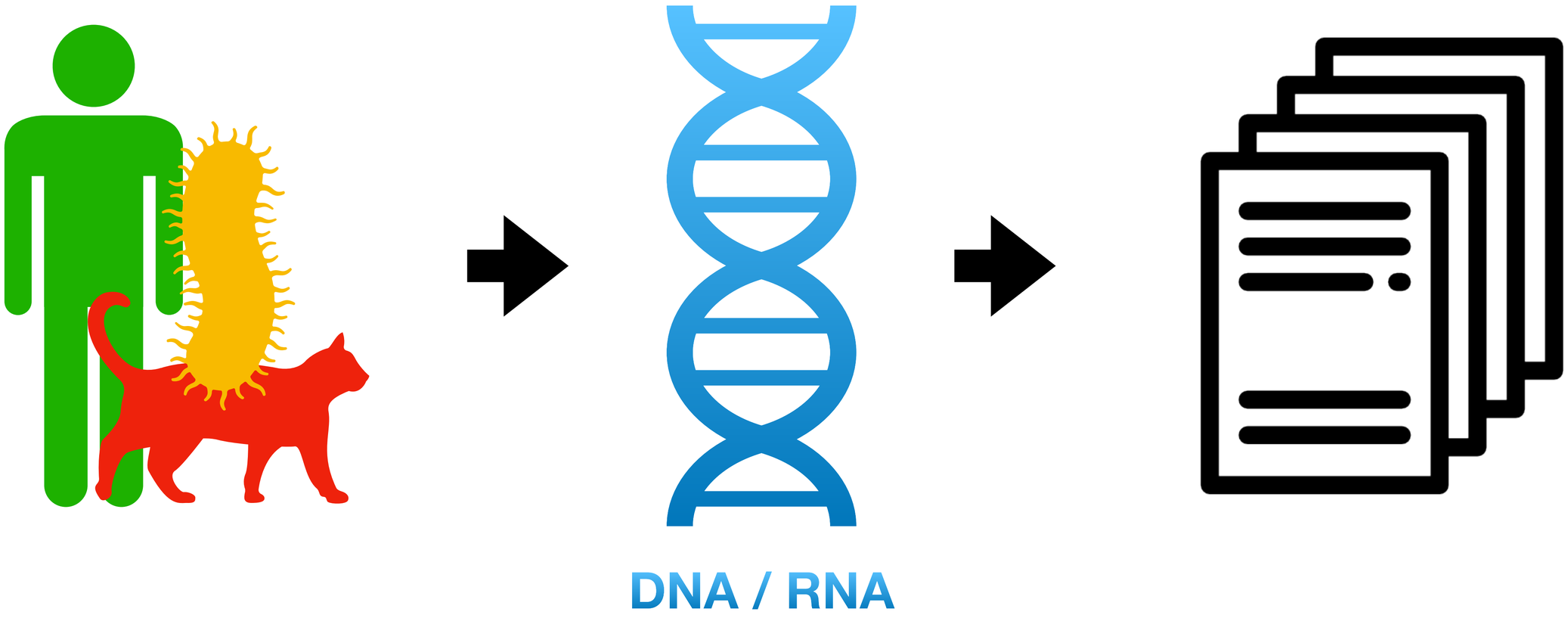
UVIC

Seq: a high-performance language for computational biology

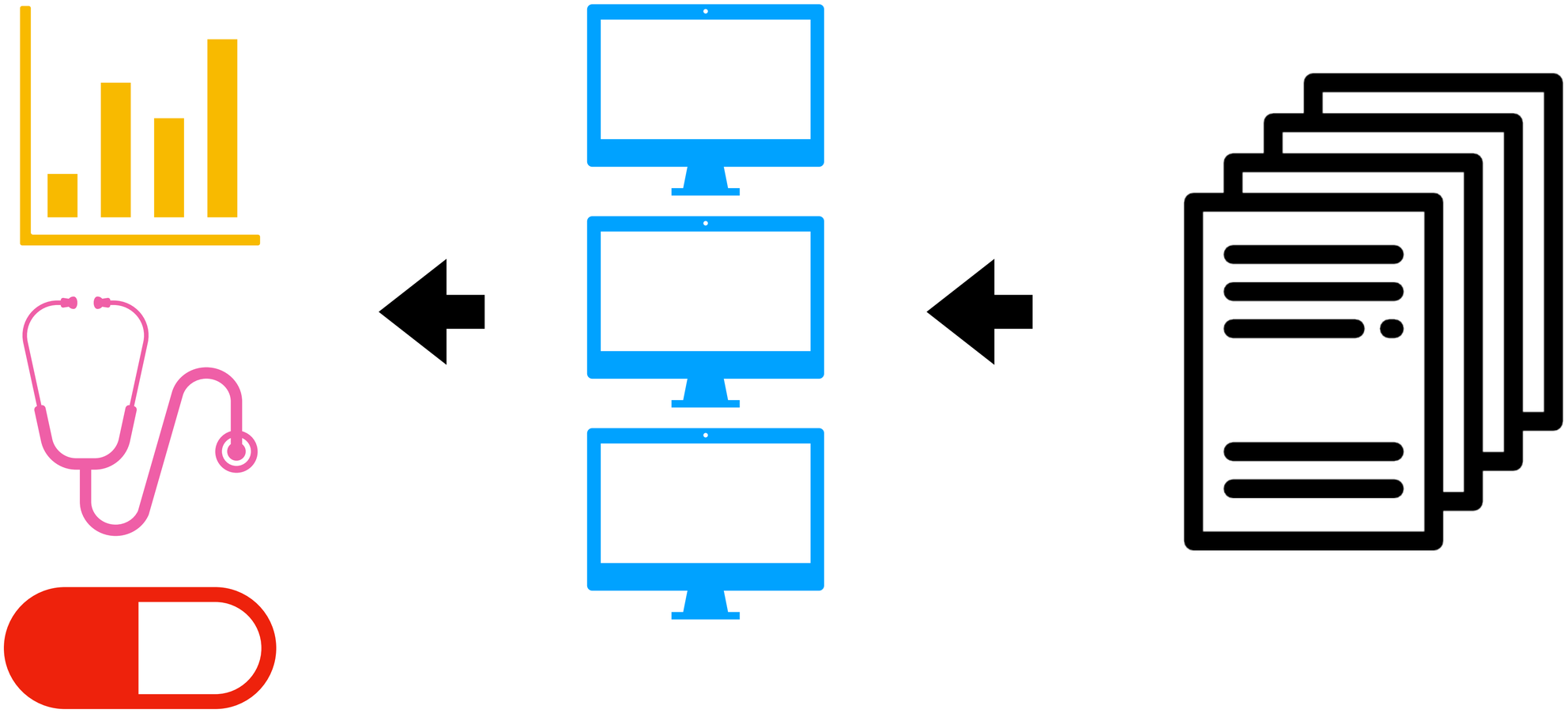
Ibrahim Numanagić
University of Victoria

Matrix Institute for Applied Data Science
1. XI 2019.

Sequencing



Sequencing



Sequencing: better than Moore’s law

- The development of next-generation sequencing technologies made sequencing fast and really cheap
  - its growth surpassed Moore’s and Kryder’s laws long ago
  - 2019: $100 to sequence a human genome

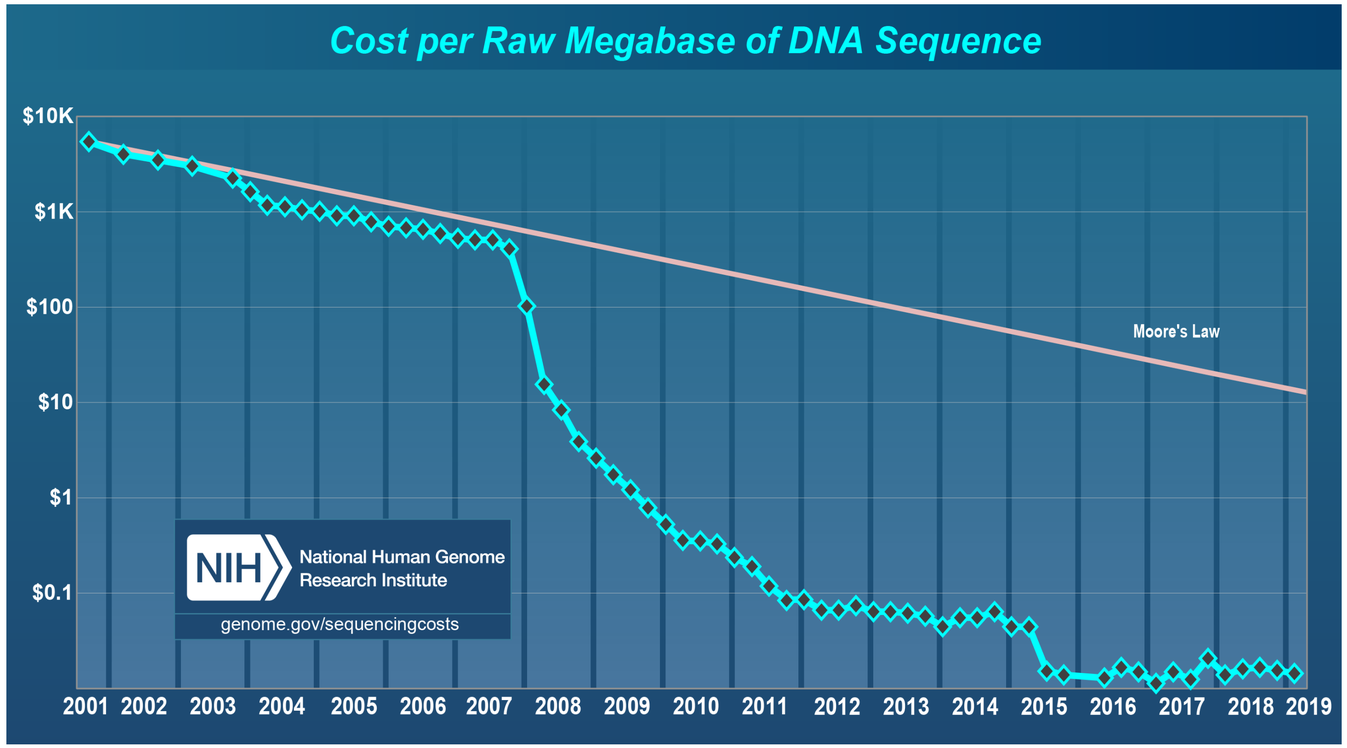


image source: NIH

Computing is the bottleneck

- But the computing did not scale as fast
- The biggest bottleneck in sequencing pipelines today:

computational data analysis

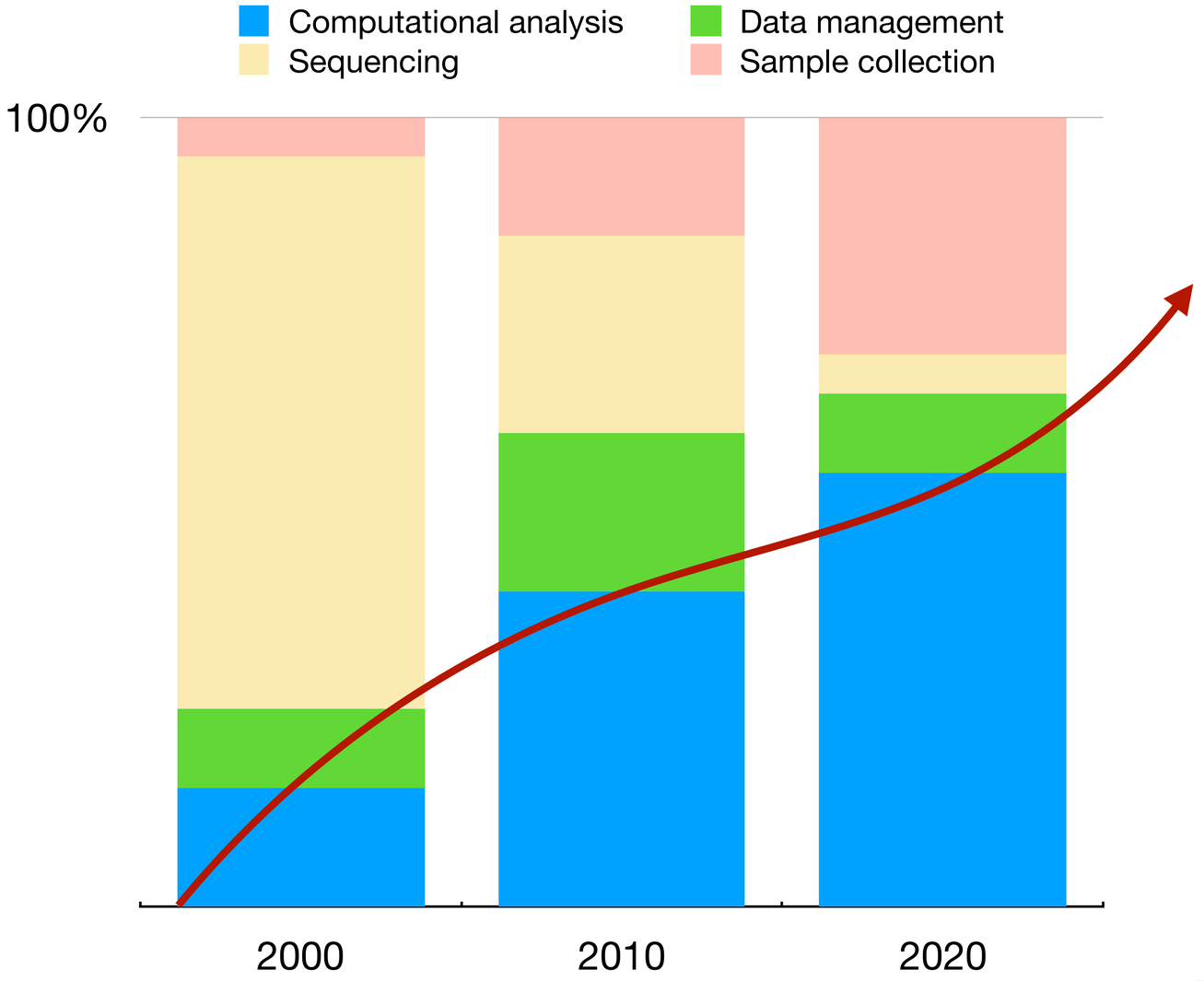
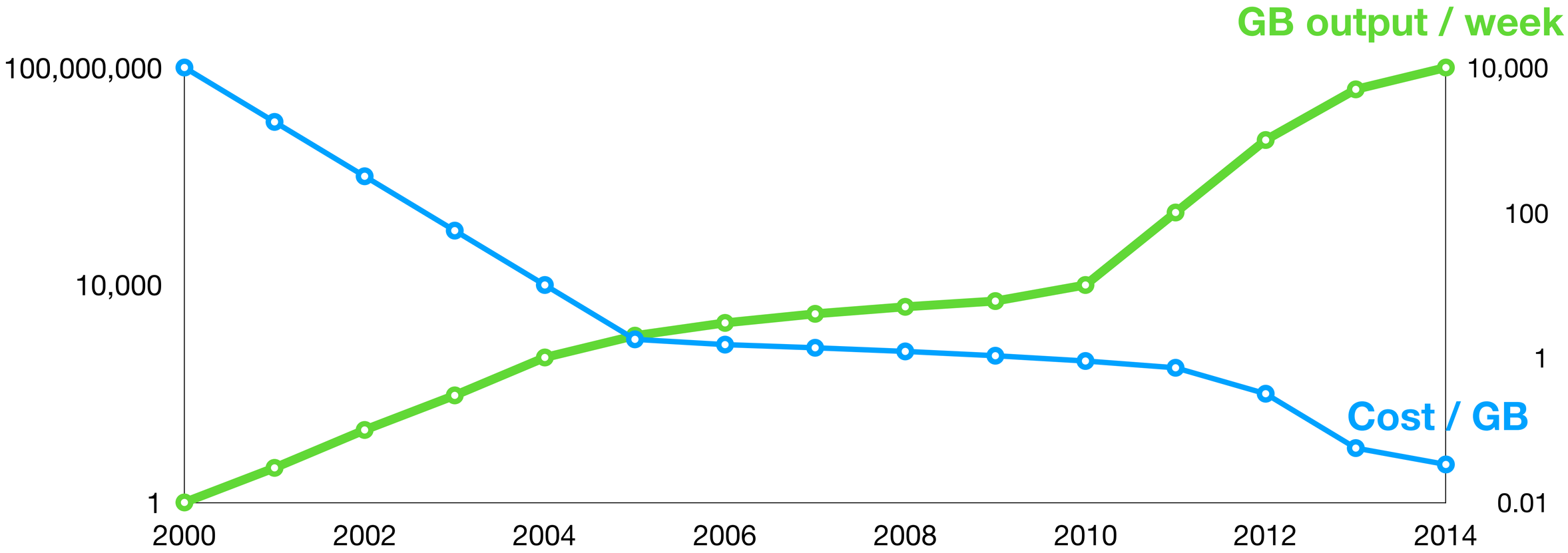


image source: Sboner *et al*. *Genome Biology* 2011, **12**:125

Reason #1: Enormous scale of data



A single sequencing experiment can generate ½ TB of data nowadays!

image source: Illumina Inc.

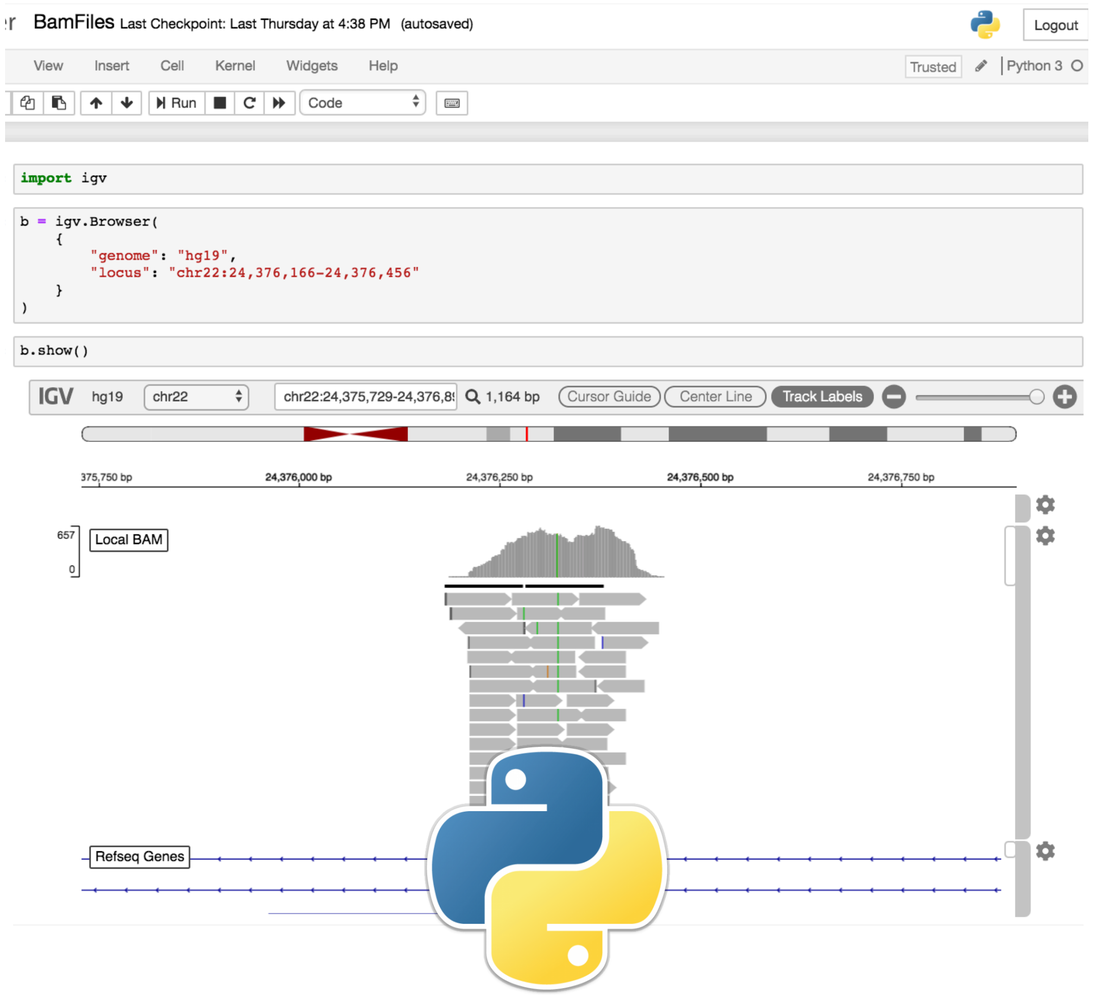
Reason #2: Rapid technology changes

- Too many platforms to support
- Soon, most of these will be obsolete anyway…

image source: Illumina Inc.

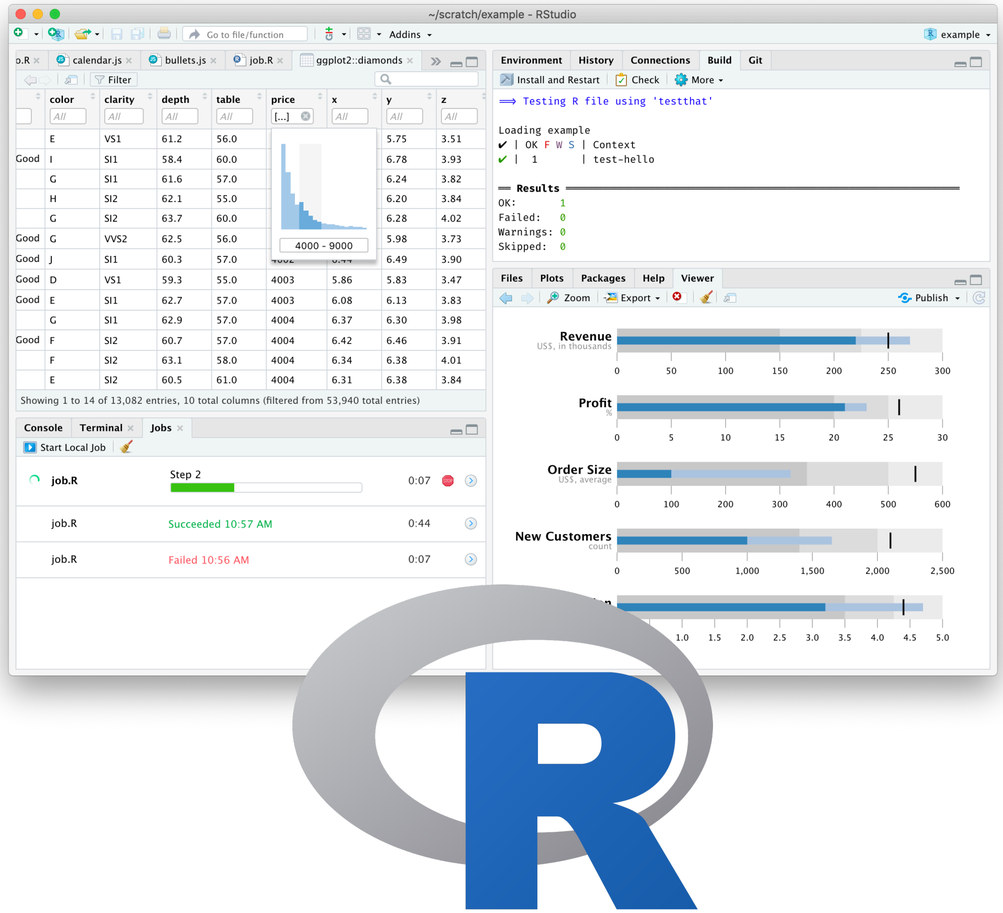
Outdated development practices

- Two camps of developers depending on the language used:
  - ***Accessible languages:*** popular, easy to develop and easy to understand **but** too slow (Python, R)



Outdated development practices

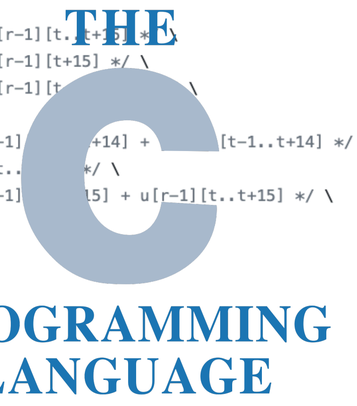
- Two camps of developers depending on the language used:
  - ***Accessible languages:*** popular, easy to develop and easy to understand **but** too slow (Python, R)



Outdated development practices

- Two camps of developers depending on the language used:
  - ***Accessible languages:*** popular, easy to develop and easy to understand **but** too slow (Python, R)
  - ***Fast languages***: fast… **but** hard and cumbersome to develop and maintain (C, C++)

```
#ifdef KSW_SSE2_ONLY
#undef __SSE4_1__
#endif

#ifdef __SSE4_1__
#include <smmintrin.h>
#endif

#ifdef KSW_CPU_DISPATCH
#ifdef __SSE4_1__
void ksw_extd2_sse41(void *km, int qlen, const uint8_t *query, int tlen, const uint8_t *target, int8_t m, con
                                    int8_t q, int8_t e, int8_t q2, int8_t e2, int w, int zdrop, int end_bonus,
#else
void ksw_extd2_sse2(void *km, int qlen, const uint8_t *query, int tlen, const uint8_t *target, int8_t m, cons
                                    int8_t q, int8_t e, int8_t q2, int8_t e2, int w, int zdrop, int end_bonus,
#endif
#else
void ksw_extd2_sse(void *km, int qlen, const uint8_t *query, int tlen, const uint8_t *target, int8_t m, const
                                    int8_t q, int8_t e, int8_t q2, int8_t e2, int w, int zdrop, int end_bonus,
#endif // ~KSW_CPU_DISPATCH
{
#define __dp_code_block1 \
        z = _mm_load_si128(&s[t]); \
        xt1 = _mm_load_si128(&x[t]);                    /* xt1 <- x[r-1][t..t+15] */ \
        tmp = _mm_srli_si128(xt1, 15);                  /* tmp <- x[r-1][t+15] */ \
        xt1 = _mm_or_si128(_mm_slli_si128(xt1, 1), x1_); /* xt1 <- x[r-1][t-1..t+14] */ \
        x1_ = tmp; \
        vt1 = _mm_load_si128(&v[t]);                    /* vt1 <- v[r-1][t..t+15] */ \
        tmp = _mm_srli_si128(vt1, 15);                  /* tmp <- v[r-1][t+15] */ \
        vt1 = _mm_or_si128(_mm_slli_si128(vt1, 1), v1_); /* vt1 <- v[r-1][t                  \
        v1_ = tmp; \
        a = _mm_add_epi8(xt1, vt1);                     /* a <- x[r-1]      +14] +      [t-1..t+14] */ \
        ut = _mm_load_si128(&u[t]);                     /* ut <- u[t..      */ \
        b = _mm_add_epi8(_mm_load_si128(&y[t]), ut);    /* b <- y[r-1]      15] + u[r-1][t..t+15] */ \
        x2t1= _mm_load_si128(&x2[t]); \
        tmp = _mm_srli_si128(x2t1, 15); \
        x2t1= _mm_or_si128(_mm_slli_si128(x2t1, 1), x21_); \
        x21_= tmp; \
        a2= _mm_add_epi8(x2t1, vt1); \
        b2= _mm_add_epi8(_mm_load_si128(&y2[t]), ut);
```



What do people use?

Anything bio-specific?

- **No language that specifically targets bioinformatics constructs**
  - Libraries are either absent or don’t cut it
  - Constant reimplementations needed for each new iteration in sequencing technologies

PacBio

MinION

Ion torrent PGM

Illumina MiSeq

Illumina NextSeq

Illumina HiSeq

A typical example of a pipeline

Bash-based DSL (why?) → Python processor

```
#
# Copyright (c) 2015 10X Genomics, Inc. All rights reserved.
#
@include "_preflight_stages.mro"
@include "_sort_fastq_by_barcode.mro"
@include "_basic_stages.mro"
@include "_aligner_stages.mro"
@include "_reporter_stages.mro"
@include "_fastq_prep_stages.mro"

pipeline BASIC(
    in  string   fastq_mode          "configuration of the inp
    in  string   sample_id,
    in  map[]    sample_def,
    in  map      downsample,
    in  string   output_format,
    in  string   read_group,
    in  int      trim_length,
    in  string   barcode_whitelist   "name of barcode whitelis
    out fastq.gz barcoded,
    out bam      barcoded_unalig
    out csv      summary_cs,
    out json     barcode_count,
)
{
    call local preflight BASIC_P
        sample_id         = self
        read_group        = self
        output_format     = self
        sample_def        = self
        check_executables = false,
    )

    call preflight BASIC_PREFLIGHT(
        sample_id         = self.sample_id,
        read_group        = self.read_group,
        output_format     = self.output_format,
        sample_def        = self.sample_def,
        check_executables = true,
    )
```

```
#!/usr/bin/env python
#
# Copyright (c) 2015 10X Genomics, Inc. All rights reserved.
#
import os
import re
import socket
import martian
import tenkit.preflight as tk_preflight

__MRO__ = """
stage ALIGNER_PREFLIGHT(
"""

def main(args, outs):
    hostname = socket.gethostname()

    if args.sample_id is not None:
        if not re.match("^[\w-]+$", args.sample_id):
            martian.exit("Sample name may only contain letters, numbers, underscores, and

    for sample_def in args.sample_def:
        read_path = sample_def["read_path"]
        if not read_path.startswith('/'):
            martian.exit("Specified FASTQ folder must be an absolute path: %s" % read_pat
        if not os.path.exists(read_path):
            martian.exit("On machine: %s, specified FASTQ folder does not exist: %s" % (h
```

then a pinch of Go

```
import (
        "encoding/binary"
        "fmt"
        "os"

        "commonFunc"
        "github.com/biogo/hts/bam"
        "github.com/biogo/hts/sam"
)

var (
        //MaxMolLen int = 1000000
        MinGap int = 1000 // minimum gap required to
)

rom(file, chrom string) {
:= os.Open(file)
nc.CheckErr(err0)
Close()

ights reserved.
```

and bit of

All this for simple

read(dna) |> correct |> split |> align |> collect

dash of Rust
(everything is re-implemented
of course)

```
use ndarray::{Array};
use std::f32;
use std::f64;
use statrs::distribution::{ChiSquared, Univariate, Continuous};

fn log_sum_exp(p: &Vec<f64>) -> f64{
    let max_p: f64 = p.iter().cloned().fold(f64::NEG_INFINITY, f64::max);
    let sum_rst: f64 = p.iter().map(|x| (x - max_p).exp()).sum();
    max_p + sum_rst.ln()
}

pub struct SmoothingInfo {
    pub num_states:      usize,
    pub num_positions:   usize,
    pub forward_prob:    Vec<Vec<f64>>,
    pub backward_prob:   Vec<Vec<f64>>,
    pub status_prob:     Vec<Vec<f64>>, // prob(x_n | o_1, o_2, ....., o_N)
    pub null_state:      usize,
    pub compared_to_second_best:    bool,
```

A curious example of a pipeline

```
makefile_location:= $(word  $(words $(MAKEFILE_LIST)), $(MAKEFILE_LIST))
makefile_realpath:= $(shell realpath $(makefile_location))
BIN:=               $(shell dirname  $(makefile_realpath) )

include $(BIN)/../NCP.common
include ../project.settings
##############################################


informative.clouds: ../plot/informative.clouds ../plot/.success
   cat $< | grep -P "^$(CHR)\t" > $@.tmp && mv $@.tmp $@

#concatenation of the real and WGS clouds
#../wgs-clouds/wgs.clouds
combined.clouds: informative.clouds $(WGS_CLOUDS)
   $(CT)
   cat $^ | subsample.pl $(REFHAP_SUBSAMPLE) > $@.tmp && mv $@.tmp $@

#chromosome intervals
intervals.txt: combined.clouds
   $(CT)
   $(BIN)/clouds2intervals.pl --min-size=2000000 < $< > $@.tmp && mv $@.tmp $@


INTERVALS:=$(shell perl -e 'print join(" ",1..`cat intervals.txt | wc -l `)')

INT=$(shell cat intervals.txt | head -n $* | tail -n 1)
$(INTERVALS:=.clouds): %.clouds: combined.clouds intervals.txt
   $(CT)
   cat $< | $(BIN)/select_by_interval.pl $(INT) > $@.tmp && mv $@.tmp $@


$(INTERVALS:=.full.cld): %.full.cld: %.clouds %.het.snp.vcf %.snp.hap %.clouds.blocks
   $(CT)
   [ -e $(JAVA_BIN) ]
   cat $< | $(BIN)/clouds2refhap.pl --chr $(CHR) --vcf $*.het.snp.vcf --dict $*.dictionary.txt > $*.frags
   $(JAVA_BIN) -cp $(BIN)/SIH.jar mpg.molgen.sih.main.SIH $*.frags $*.refhap_phase
```

- **Makefile-driven development**
  - whole software package is a collection of Makefiles
  - This is *industrial grade* bioinformatics software… that almost nobody uses!

Can we make it better?

A genomics primer: building blocks

- Computational genomics applications use the same set of core operations:
  - String manipulation on a limited alphabet (typically A, C, G, T and N)
  - frequent genome queries and index lookups
  - dynamic programming algorithms such as string alignment

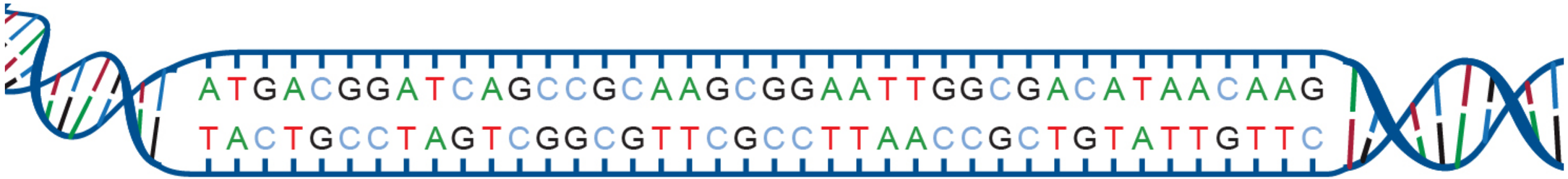
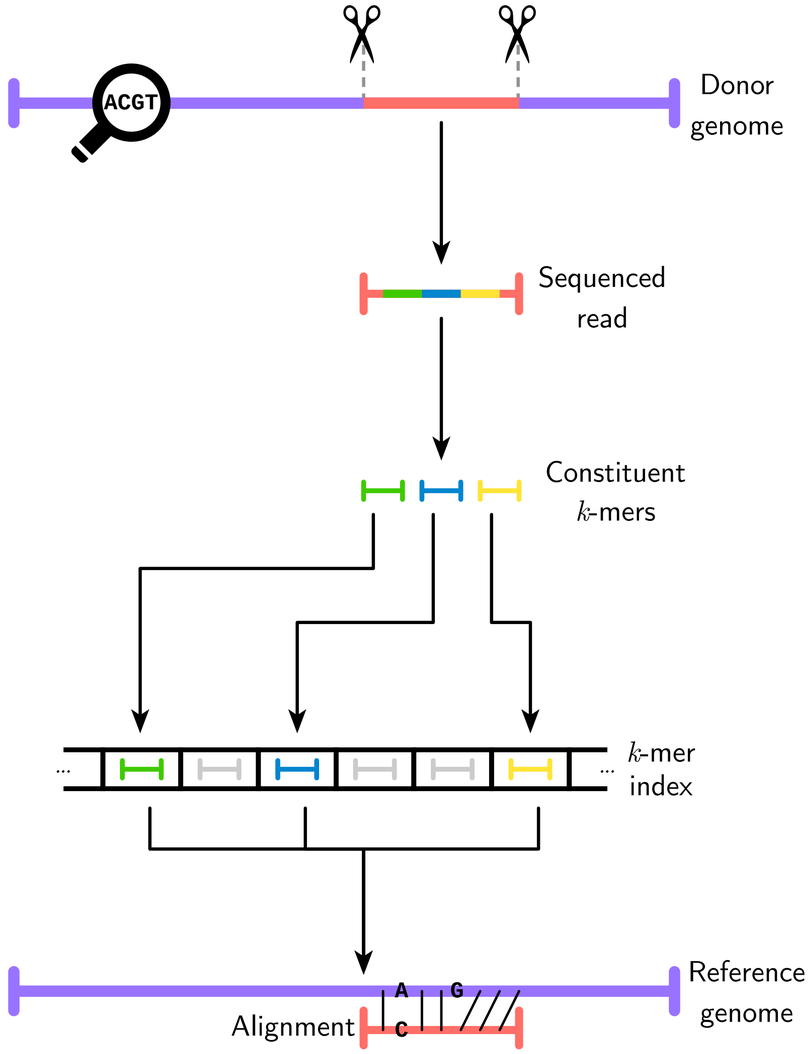


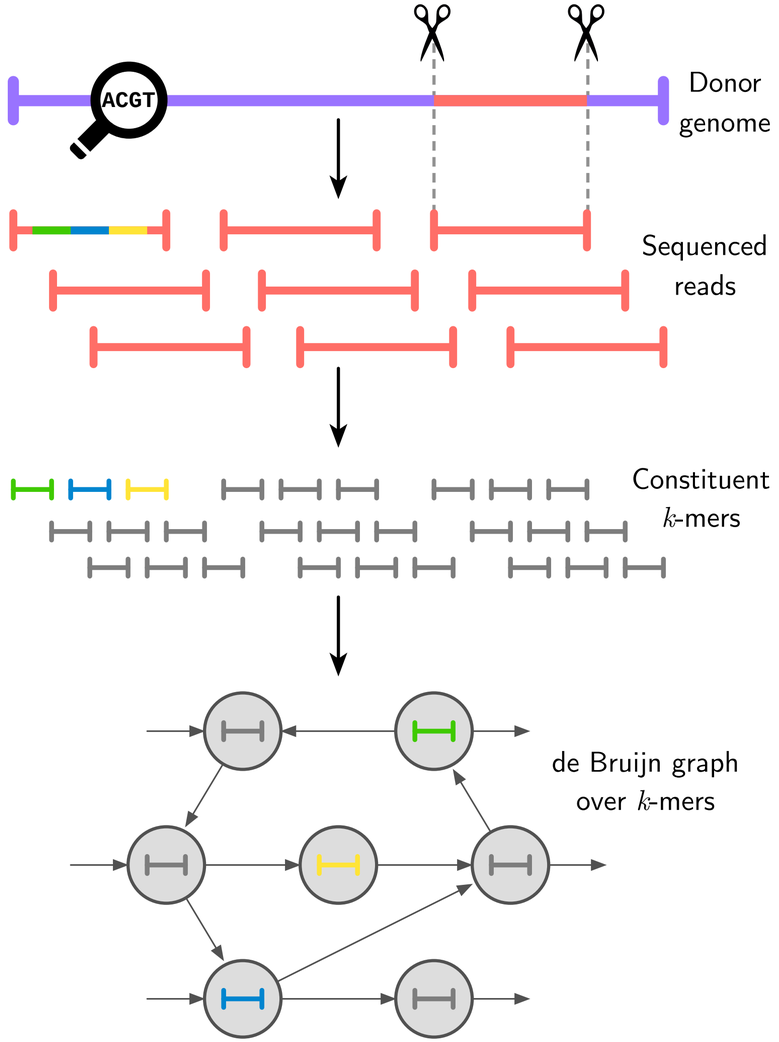
image source: genome.gov

A genomics primer: alignment



- Each sequencing machine produces reads: short DNA sequences of size ~100
- **Read mapping**: find the origin location of a read in the reference genome
    1. Split a read into *k*-mers: small fixed length-*k* subsequences)
    2. Find the occurrences of each *k*-mer in the reference genome by querying the genome index
    3. Run dynamic programming to produce the final alignment
    4. Repeat this for 100,000,000+ reads

A genomics primer: assembly



- **Read assembly**: reconstruct the reference genome from sequenced reads
  - Make a de Bruijn graph from read *k*-mers
    - its edges represent (*k* – 1)-length overlaps between the nodes (*k*-mers)
  - ***Assembly contig***: an Eulerian path in de Bruijn graph
- NP-hard due to genome repeats

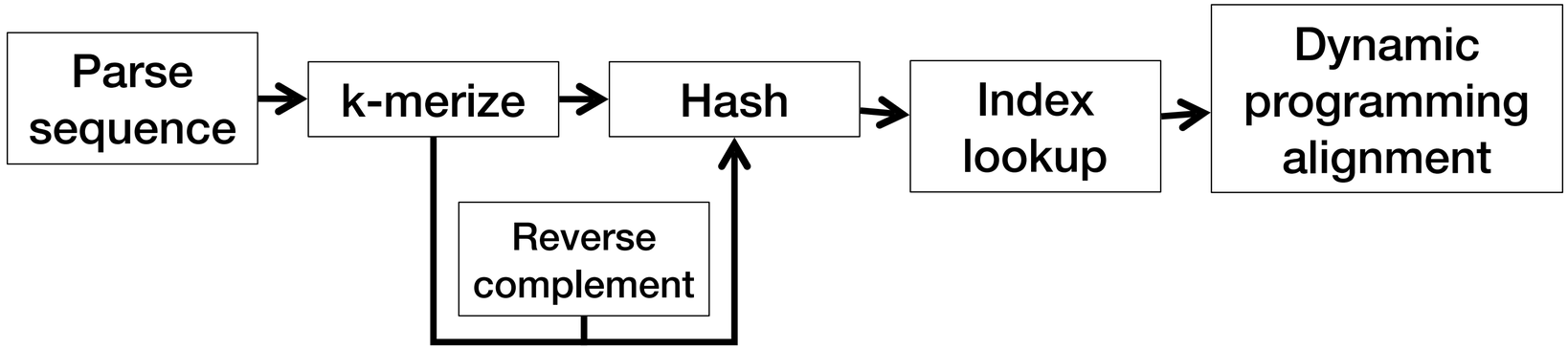
How about domain-specific language?

- **Won't work!**
  - Computational biology *data* has general laws, but…
  - … the target *computational* domain is **too general**
  - Any algorithm or a data structure out there— we have it and we use it

- **Example**: EMA aligner for third-generation barcoded sequencing technologies
  - Large file processing (splitting, sorting, I/O)
  - Alignment and other pattern matching methods
  - Probabilistic methods (EM, simulated annealing)
  - Integer linear programming

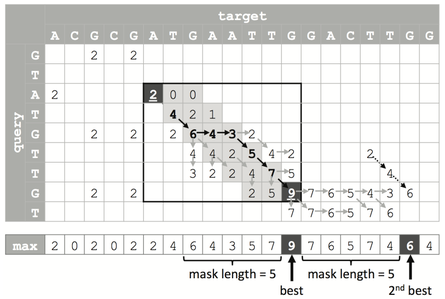
We need a two-tier approach

- **Top-down *(high-level)***: describe the problem intuitively without thinking about optimizations



- **Bottom-up *(low-level)***: implement high-performant and scalable components

=



MIT CSAIL

understand genomics

general purpose

fast and scalable

easy & rapid development

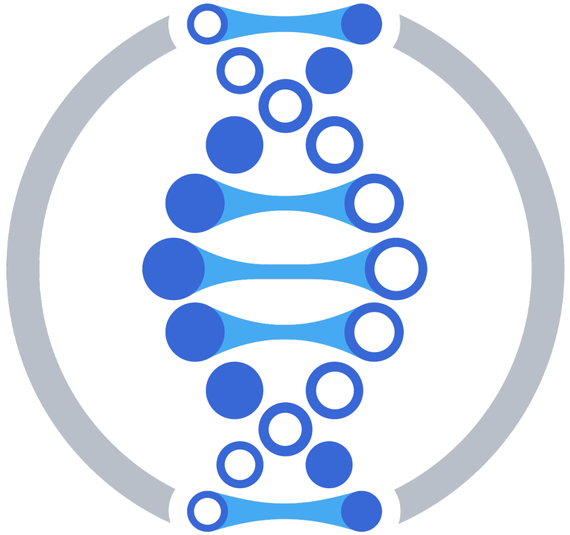
understand genomics

general purpose

fast and scalable

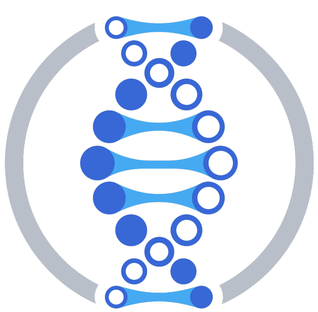
easy & rapid development

=



Seq

Seq: a language for computational biology



Seq

=

Ease of Python

+

Performance of C

+

Genomics features

- Our approach: **Seq**, a general language with a host of genomics-related features and optimizations

Seq: a primer

C++

```cpp
#include <iostream>
#include <fstream>
#include <string>
#include <cstdlib>
#include "GenomeIndex.h"

char revcomp(char base) {
  switch (base) {
    case 'A': return 'T';
    case 'C': return 'G';
    case 'G': return 'C';
    case 'T': return 'A';
    default: return base;
  }
}

void revcomp(char *kmer, int k) {
  for (int i = 0; i < k/2; i++) {
    char a = revcomp(kmer[i]);
    char b = revcomp(kmer[k - i - 1]);
    kmer[i] = b;
    kmer[k - i - 1] = a;
  }
}
```

```cpp
void process(char *kmer, int k,
             GenomeIndex &index) {
  auto hits_fwd = index[kmer];
  revcomp(kmer, k);
  auto hits_rev = index[kmer];
  revcomp(kmer, k);  // undo
  ...
}

int main(int argc, char *argv[]) {
  const int k = 20;
  const int stride = 10;
  auto *index = GenomeIndex(argv[1], k);
  std::ifstream fin(argv[2]);
  std::string read;
  long line = -1;
  while (std::getline(fin, read)) {
    line++;
    // skip over non-sequences in FASTQ
    if (line % 4 != 1) continue;
    auto *buf = (char *)read.c_str();
    int len = read.size();
    for (int i = 0; i + k <= len; i += stride)
      process(kmer, k, index);
  }
}
```

vs.

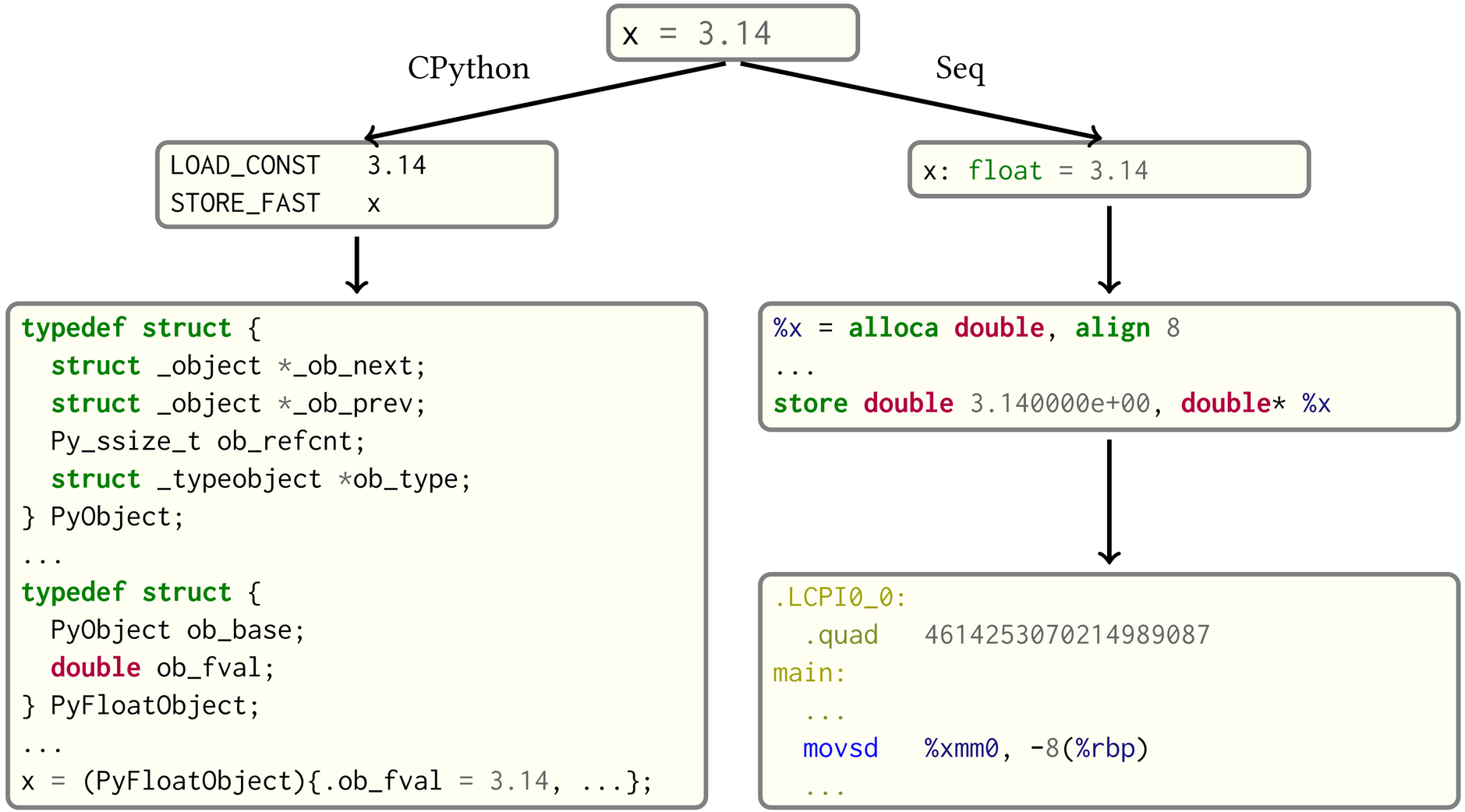
Seq

```
from sys import argv
from genomeindex import *
type K = Kmer[20]

# index and process 20-mers
def process(kmer: K,
            index: GenomeIndex[K]):
    prefetch index[kmer], index[~kmer]
    hits_fwd = index[kmer]
    hits_rev = index[~kmer]
    ...

# index over 20-mers
index = GenomeIndex[K](argv[1])

# stride for k-merization
stride = 10

# sequence-processing pipeline
(fastq(argv[2])
  |> kmers[K](stride)
  |> process(index))
```

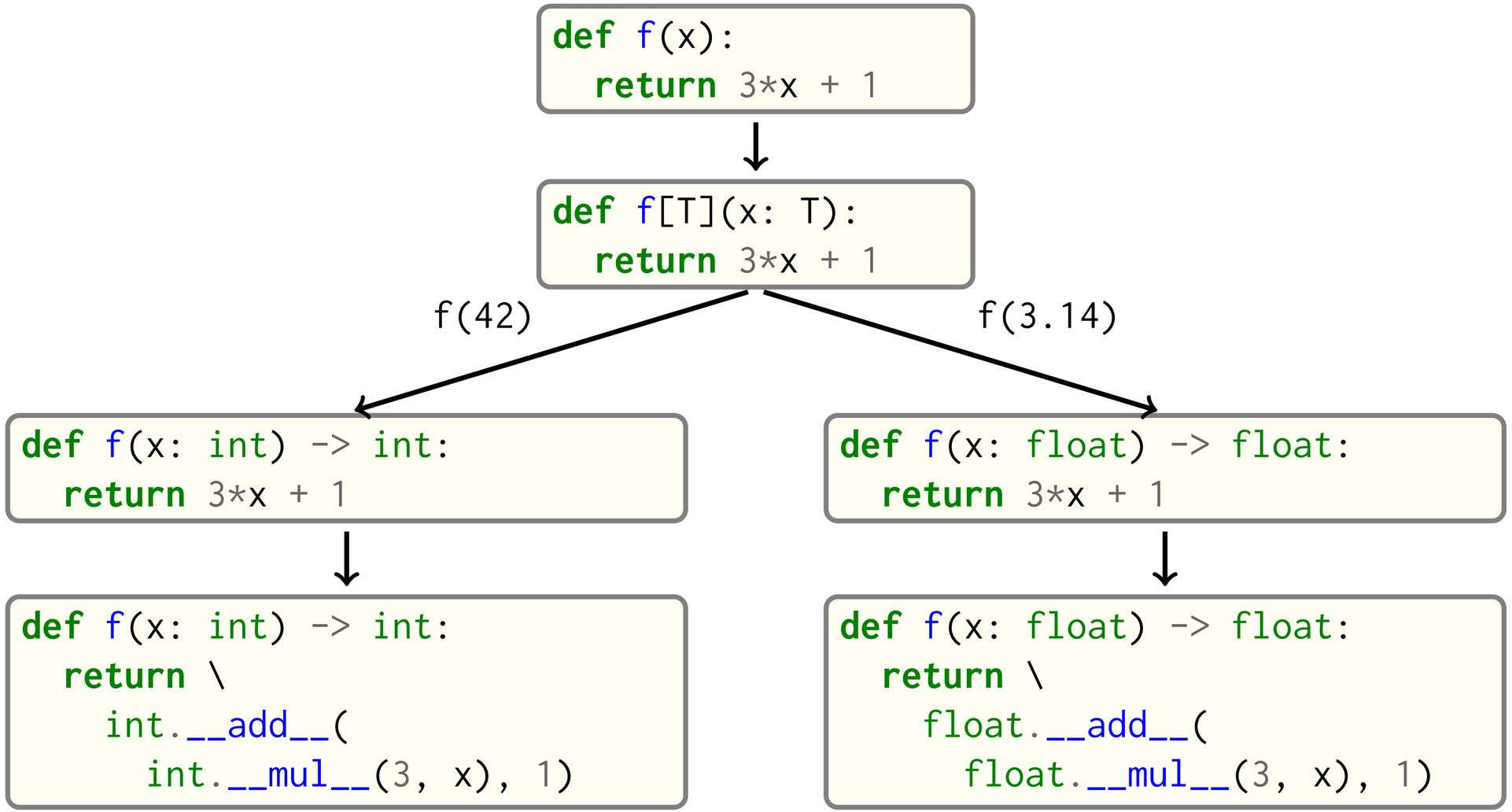
Implementation: a strongly-typed Python

- Python is duck-typed dynamic language that completely relies on runtime
- We want the **syntax and clarity of Python** with **none of the runtime overhead**
  - Most of the Python's overhead stems from its dynamic runtime capabilities that are rarely, if at all, used in genomics pipelines
- Solution: **strongly typed language with Python syntax that does everything at compile time**

Implementation: a strongly-typed Python



Implementation: a strongly-typed Python



Implementation: a strongly-typed Python

```
class Node[T]:
    next: Node[T]
    data: T

def item[T,U](n: Node[T], f: function[T,U]) -> list[U]:
    return [f(n.data)]

n = Node(None, 5)
def foo(x: int) -> str:
    return str(x)
i = item(n, foo)  # type parameters deduced as int and str
i = item[int,str](n, foo)  # explicit specification also OK
```

- Explicit (but optional) generics for easier type inference
- Hindley-Milner inference is being merged into Seq

Implementation: bootstrapping

- The standard library and most Pythonic constructs are bootstrapped directly in Seq
  - list[T] or dict[K,V]: all implemented in Seq
  - all functions (map, zip, sort etc.) are implemented in Seq

```
def all(x):
    for a in x:
        if not a:
            return False
    return True

def enum(x):
    i = 0
    for a in x:
        yield (i,a)
        i += 1

def zip(a, b):
    bi = iter(b)
    for i in a:
        if bi.done(): break
        yield (i, bi.next())
    bi.destroy()

def filter(f, x):
    for a in x:
        if f(a):
            yield a
```

```
class list[T]:
    arr: array[T]
    len: int

    def __init__(self: list[T], arr: array[T], len: int):
        self.arr = arr
        self.len = len

    def __len__(self: list[T]):
        return self.len

    def __bool__(self: list[T]):
        return len(self) > 0

    def __getitem__(self: list[T], idx: int):
        if idx < 0:
            idx += len(self)
        self._idx_check(idx, "list index out of range")
        return self.arr[idx]
```

Differences with Python

- **Functions can only return objects of a single type**

- ~~[42, 3.14, "hello"]~~
  Collections cannot contain objects of different types

- ~~obj.method = new_method~~
  Methods of an object cannot be modified at runtime (possible at compile time though!)

- ~~(1, 2.2)[idx]~~
  Tuple indices must be constants, and iteration over a tuple is allowed only if its elements all have the same type

- No inheritance nor polymorphism (partially alleviated by instantiation)

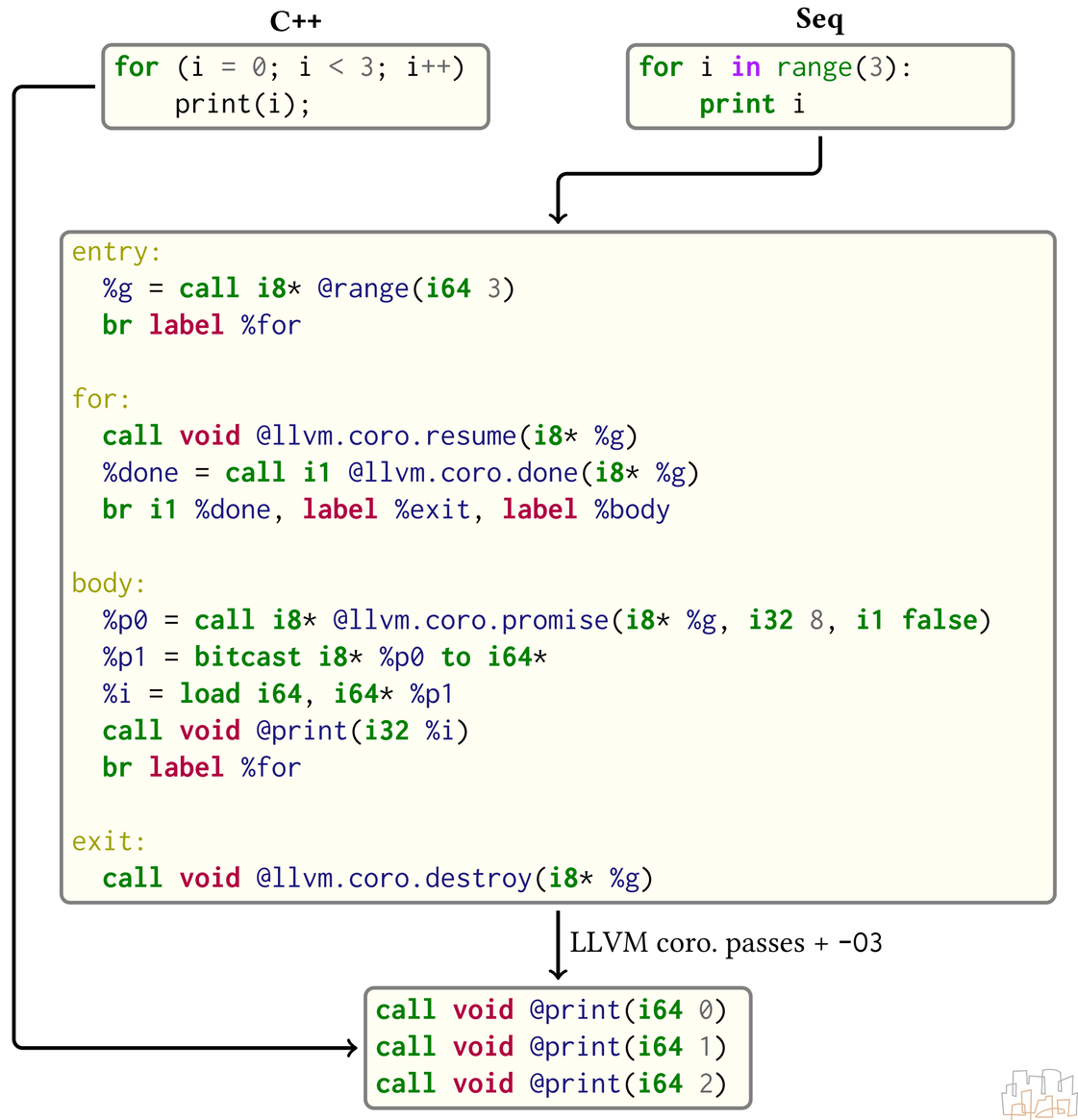
- ~~if cond():~~
  ~~ x = 1~~
  ~~else:~~
  ~~ x = 2~~
  ~~print x~~

  Stricter scoping rules than Python

- Other temporary restrictions (lambda, for-else, empty literals etc.)

Coroutines and generators

- Generators: *extremely* important feature of Python
  - for a in range(3) iterates over the generator range(3)
- **LLVM supports coroutines!**
  - Allows us to implement Python generators with virtually no overhead thanks to coroutine passes and inlining
  - Enables efficient pipelining and laziness



New features

Pipelines

Genomics types

Pattern matching

C/C++ interop

Type extensions

Prefetching

New features: pipelines

Pipelines

Genomics types

Pattern matching

C/C++ interop

Type extensions

Prefetching

```
dna = s'ACGTACGTACGT'  # sequence literal

# (a) split into subsequences of length 3
#     with a stride of 2
dna |> split(..., 3, 2) |> echo

# (b) split into 5-mers with stride 1
def f(kmer):
    print kmer
    print ~kmer

dna |> kmers[Kmer[5]](1) |> f
```

```
dna = s'ACGTACGTACGT'  # sequence literal

# (a) split into subsequences of length 3
#     with a stride of 2
dna |> split(..., 3, 2) ||> echo

# (b) split into 5-mers with stride 1
def f(kmer):
    print kmer
    print ~kmer

dna |> kmers[Kmer[5]](1) ||> f
```

|> for pipes

||> for parallel pipes

A single character difference!

New features: genomics types

Pipelines

Genomics types

Pattern matching

C/C++ interop

Type extensions

Prefetching

- **Sequence and *k*-mer types**
- **Various compile-time optimizations:**
  - 2-bit sequence encoding
  - Low-level lookup table for fast reverse complementation
  - Avoid copying and allocations unless really necessary

```
dna = s'ACGTACGTACGT'  # sequence literal

# (a) split into subsequences of length 3
#     with a stride of 2
for sub in dna.split(3, 2):
    print sub

# (b) split into 5-mers with stride 1
for kmer in dna.kmers[Kmer[5]](1):
    print kmer
    print ~kmer  # reverse complement

# (c) convert entire sequence to 12-mer
kmer = Kmer[12](dna)
```

New features: pattern matching

- Pipelines
- Genomics types
- **Pattern matching**
- C/C++ interop
- Type extensions
- Prefetching

```
def describe(n: int):
    match n:
        case m if m < 0:
            print 'negative'
        case 0:
            print 'zero'
        case m if 0 < m < 10:
            print 'small'
        default:
            print 'large'
```

New features: pattern matching

- Pipelines
- Genomics types
- **Pattern matching**
- C/C++ interop
- Type extensions
- Prefetching

```
# (a)
def has_spaced_acgt(s: seq) -> bool:
    match s:
        case s'A_C_G_T...':
            return True
        case t if len(t) >= 8:
            return has_spaced_acgt(s[1:])
        default:
            return False

# (b)
def is_own_revcomp(s: seq) -> bool:
    match s:
        case s'A...T' or s'T...A' or s'C...G' or s'G...C':
            return is_own_revcomp(s[1:-1])
        case s'':
            return True
        default:
            return False
```

```
# (c)
type BaseCount(A: int, C: int, G: int, T: int):
    def __add__(self: BaseCount, other: BaseCount):
        a1, c1, g1, t1 = self
        a2, c2, g2, t2 = other
        return (a1 + a2, c1 + c2, g1 + g2, t1 + t2)

def count_bases(s: seq) -> BaseCount:
    match s:
        case s'A...': return count_bases(s[1:]) + (1,0,0,0)
        case s'C...': return count_bases(s[1:]) + (0,1,0,0)
        case s'G...': return count_bases(s[1:]) + (0,0,1,0)
        case s'T...': return count_bases(s[1:]) + (0,0,0,1)
        default: return BaseCount(0,0,0,0)
```

New features: C interop

- Pipelines
- Genomics types
- Pattern matching
- **C/C++ interop**
- Type extensions
- Prefetching

```
cdef sqrt(float) -> float
cdef puts(ptr[byte])
print sqrt(100.0)
puts("hello world".c_str())
```

- Python and R interop capabilities (pydef and rdef) are coming soon as well

New features: type extensions

- Pipelines
- Genomics types
- Pattern matching
- C/C++ interop
- **Type extensions**
- Prefetching

```
extend int:
    def to(self: int, other: int):
        for i in range(self, other + 1):
            yield i

    def __mul__(self: int, other: int):
        print 'caught int mul!'
        return 42

for i in (5).to(10):
    print i  # 5, 6, ..., 10

# prints 'caught int mul!' then '42'
print 2 * 3
```

New features: prefetching

- Pipelines
- Genomics types
- Pattern matching
- C/C++ interop
- Type extensions
- **Prefetching**

- In many popular genomics tools, more than 50% of time is wasted on stalling

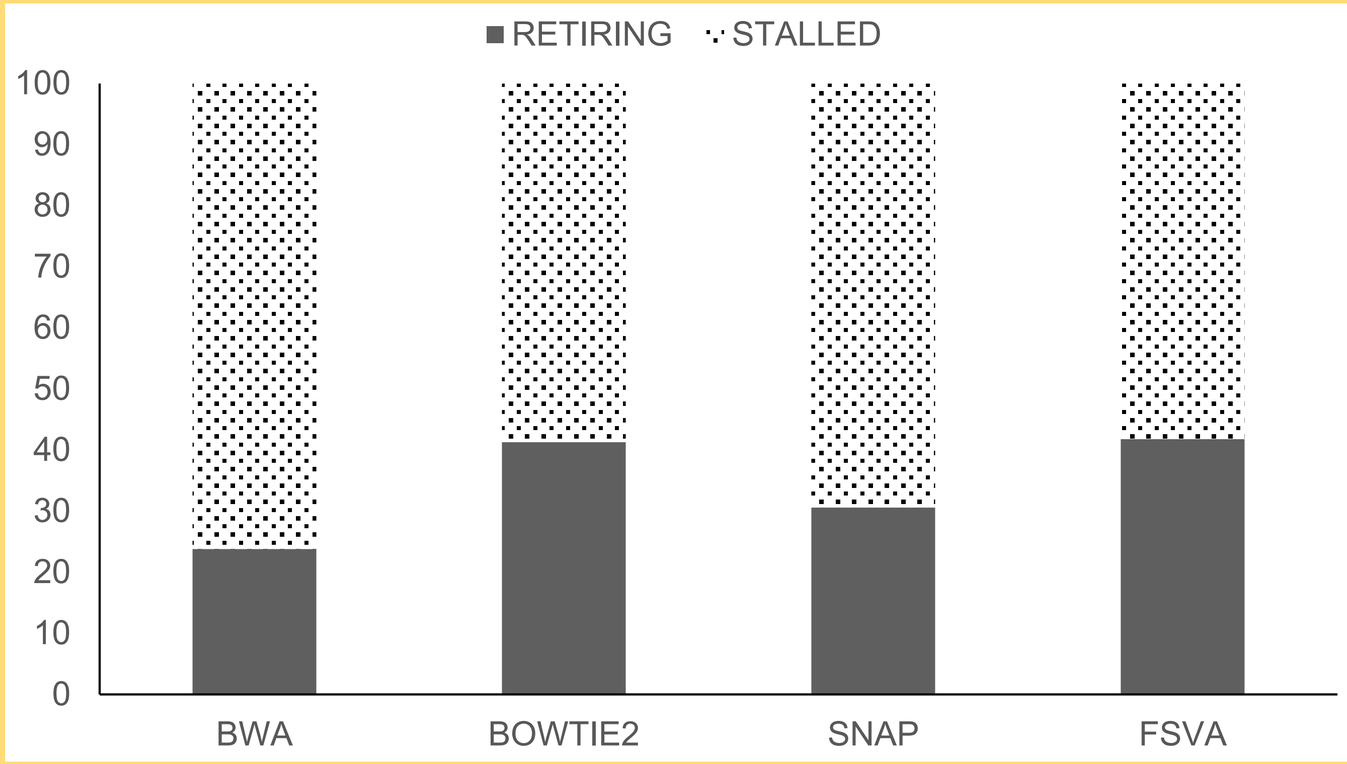


image source: Appuswamy *et al*. *IEEE IPDPSW* 2018

New features: prefetching

- Pipelines
- Genomics types
- Pattern matching
- C/C++ interop
- Type extensions
- **Prefetching**

- **Seq supports dynamic prefetching for faster index queries (*a la* Cimple[1])**

1. Just add a single prefetch statement to indicate prefetching

```
type k20 = Kmer[20]
def process(read: seq, index: MyIndex):
  ...
  for kmer in read.kmers[k20](step):
    prefetch index[kmer], index[~kmer]
    hits = index[kmer]
    hits_rev = index[~kmer]
    ...
  return x
```

2. Make sure that index exposes __prefetch__ magic

```
class MyIndex:  # abstract k-mer index
  ...
  def __getitem__(self: MyIndex, kmer: Kmer[20]):
    # standard __getitem__
  def __prefetch__(self: MyIndex, kmer: Kmer[20]):
    # similar to __getitem__, but performs prefetch
```

[1] Kiriansky *et al*. *ACM PACT* 2018

New features: prefetching

- Pipelines
- Genomics types
- Pattern matching
- C/C++ interop
- Type extensions
- **Prefetching**

- The function surrounding prefetch is automatically converted to a generator

```
type k20 = Kmer[20]
def process(read: seq, index: MyIndex):
  ...
  for kmer in read.kmers[k20](step):
    prefetch index[kmer], index[~kmer]
    hits = index[kmer]
    hits_rev = index[~kmer]
    ...
  return x
```

```
type k20 = Kmer[20]
def process(read: seq, index: MyIndex):
  ...
  for kmer in read.kmers[k20](step):
    index.__prefetch__(kmer)
    index.__prefetch__(~kmer)
    yield
    hits = index[kmer]
    hits_rev = index[~kmer]
    ...
  yield x
```

New features: prefetching

- Pipelines
- Genomics types
- Pattern matching
- C/C++ interop
- Type extensions
- **Prefetching**

```
FASTQ("reads.fq")  # input reads
 |> process(index) # index lookup
 |> postprocess    # output results
```



```
M = ...  # num. concurrent tasks
N = 0    # next coroutine slot to fill
k = 0    # next coroutine to execute
states = array[generator[T]](M)

for read in FASTQ("reads.fq"):
  if N < M:
    states[N] = process(read, index)
    N += 1
  else:
    while True:
      g = states[k]; g.next()
      if g.done():
        postprocess(g.promise())
        g.destroy()
        states[k] = process(read, index)
        break
      k = (k + 1) % M

for i in range(N):
  g = states[i]
  if not g.done():
    while not g.done(): g.next()
    postprocess(g.promise())
    g.destroy()
```

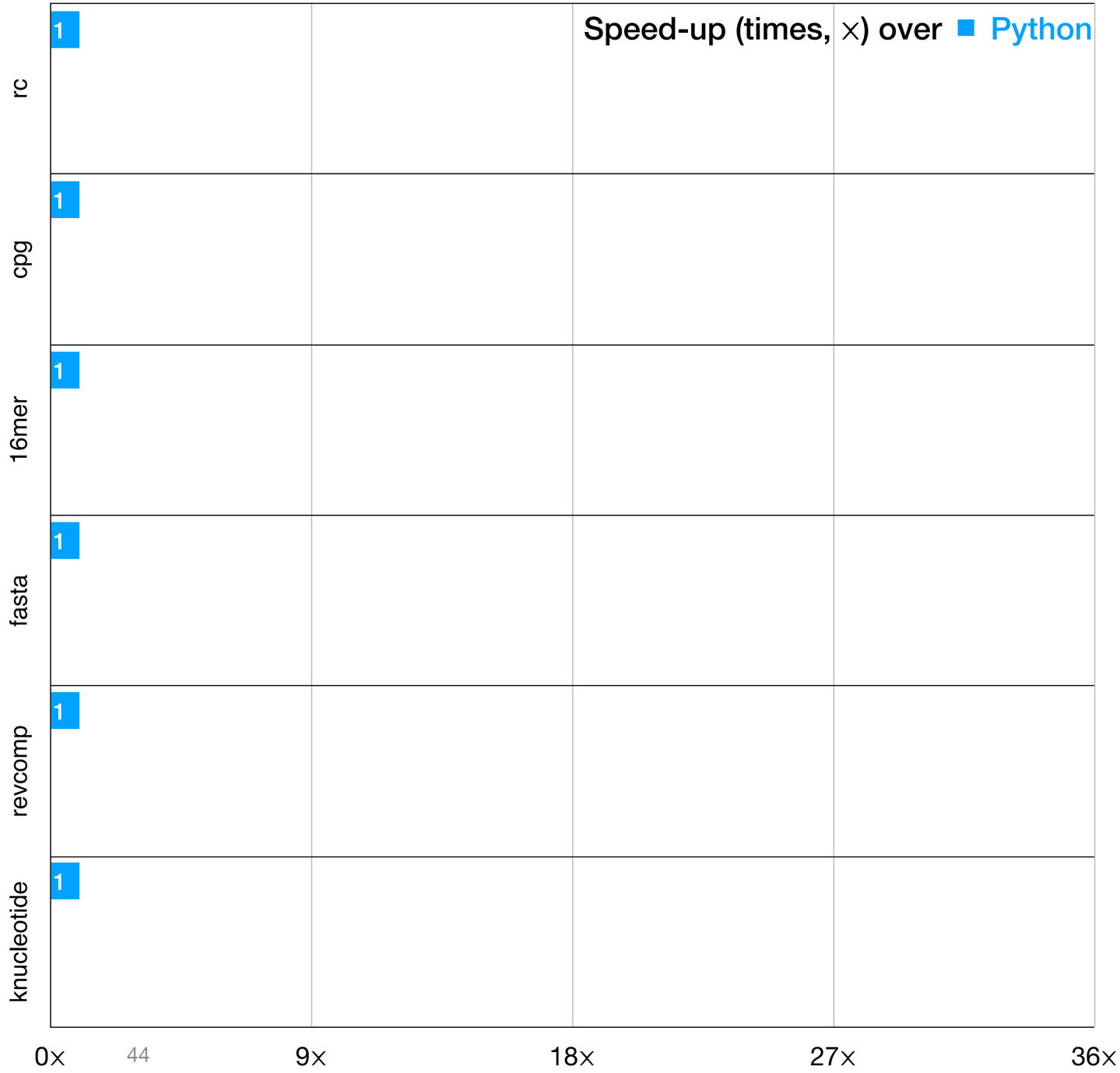
- The pipeline that uses a prefetch function is also transformed to allow suspension
- Only a single line modification
- Results in up to 50% runtime improvements over the baseline

Benchmarks

1. **Computer Language Benchmarks Game**
   i. fasta (*60 LOC, 2 min*)
   ii. revcomp (*35 LOC, 2 min*)
   iii. knucleotide (*40 LOC, 4 min*)
2. **In-house suite (100 million reads)**
   i. rc (*25 LOC, 35 min*)
   ii. 16mer (*35 LOC, 4+ hrs*)
   iii. cpg (*40 LOC, 1 hr*)
3. **Genome index suite**
   i. snap (*70 LOC, 8 min*):
   query *k*-mer-based genome index
   ii. sga (*100 LOC, 9 min*):
   query FM-based genome index

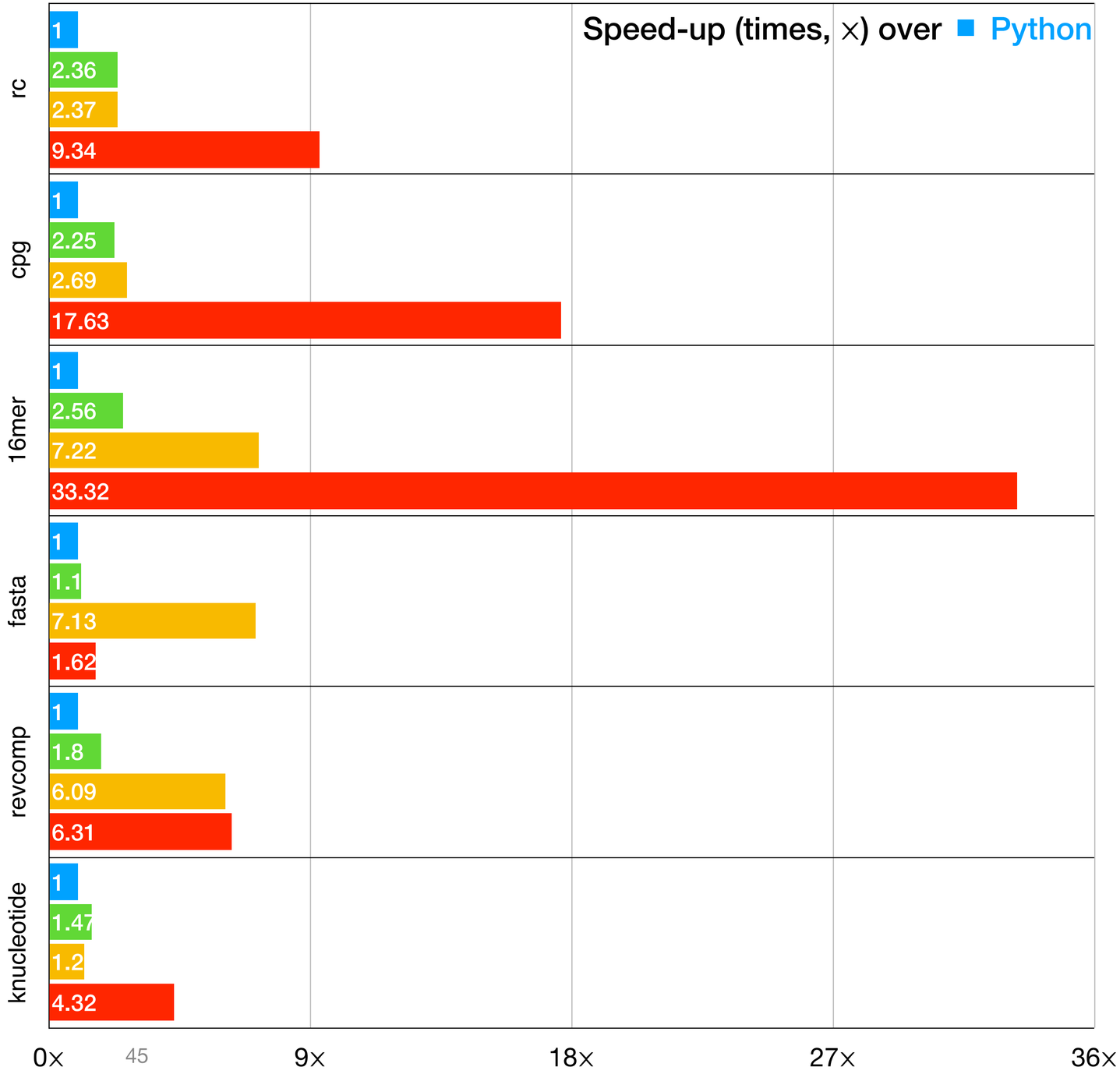
Benchmarks

- ■ **Python** is the reference implementation



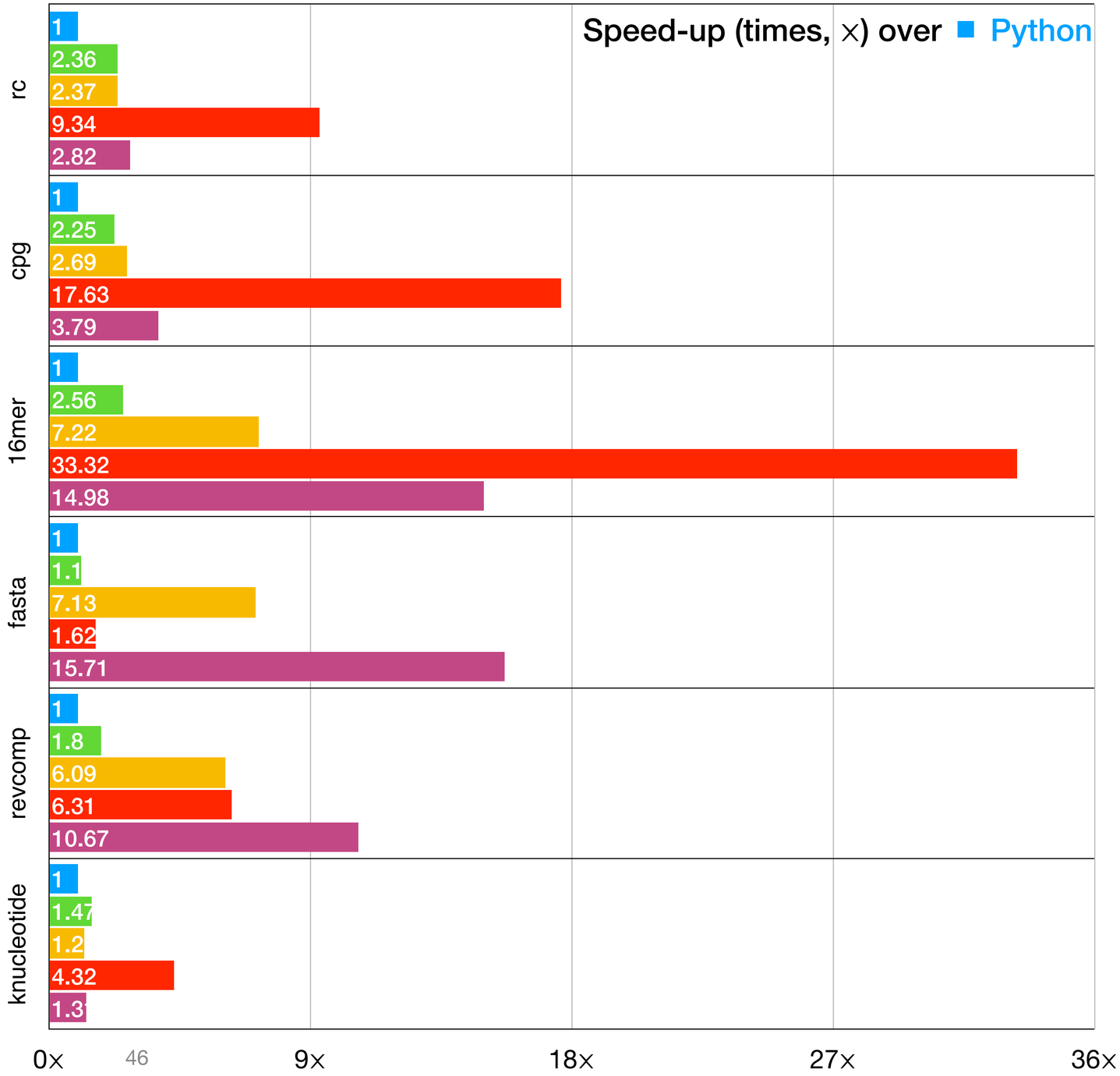
Benchmarks

- ■ **Python** is the reference implementation
- Compiled and JIT Pythons, such as ■ **Nuitka**, ■ **Shedskin** and ■ **PyPy** help a bit



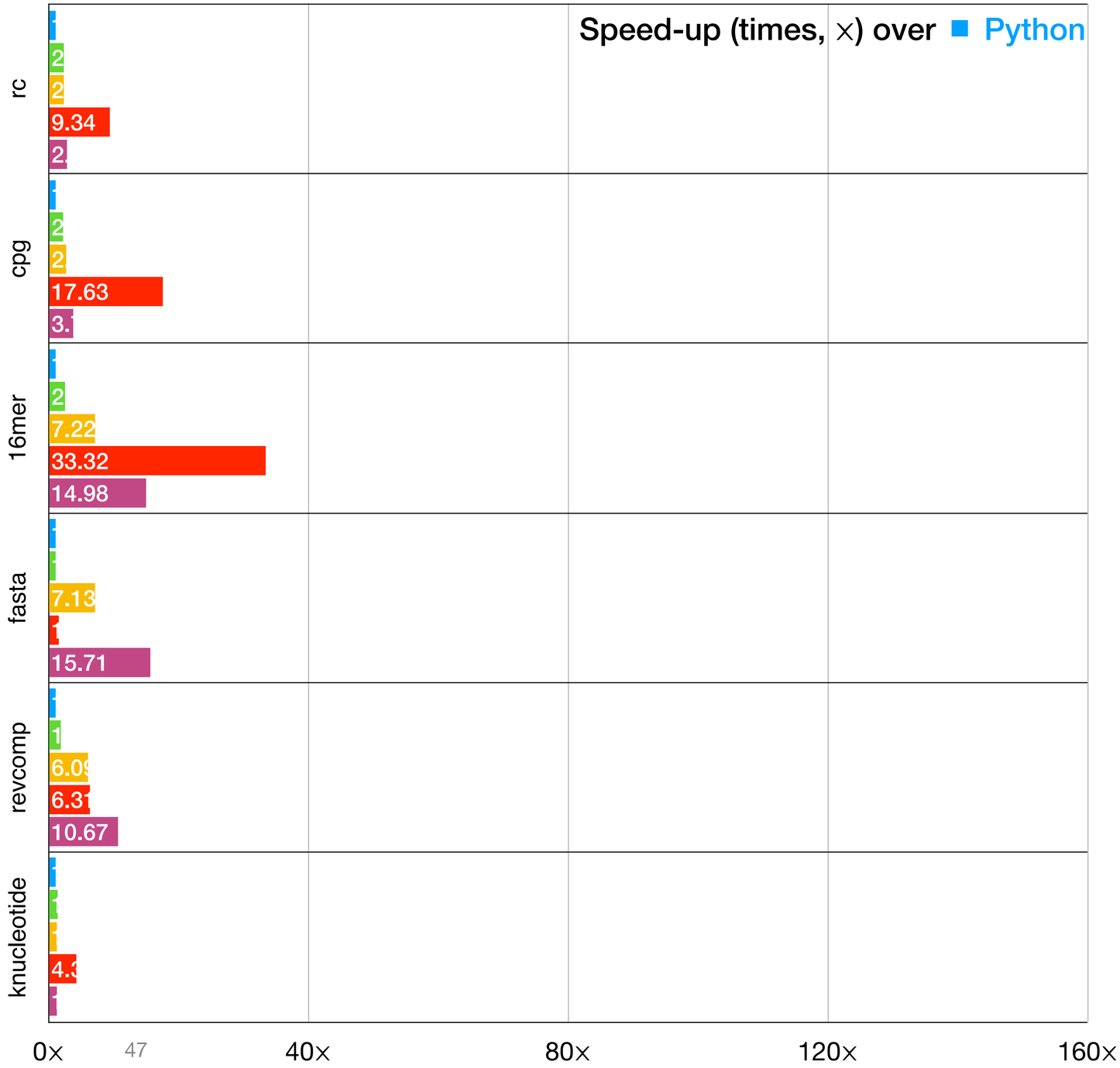
Benchmarks

- ■ **Python** is the reference implementation
- Compiled and JIT Pythons, such as ■ **Nuitka**, ■ **Shedskin** and ■ **PyPy** help a bit
- ■ **Julia** is similar…



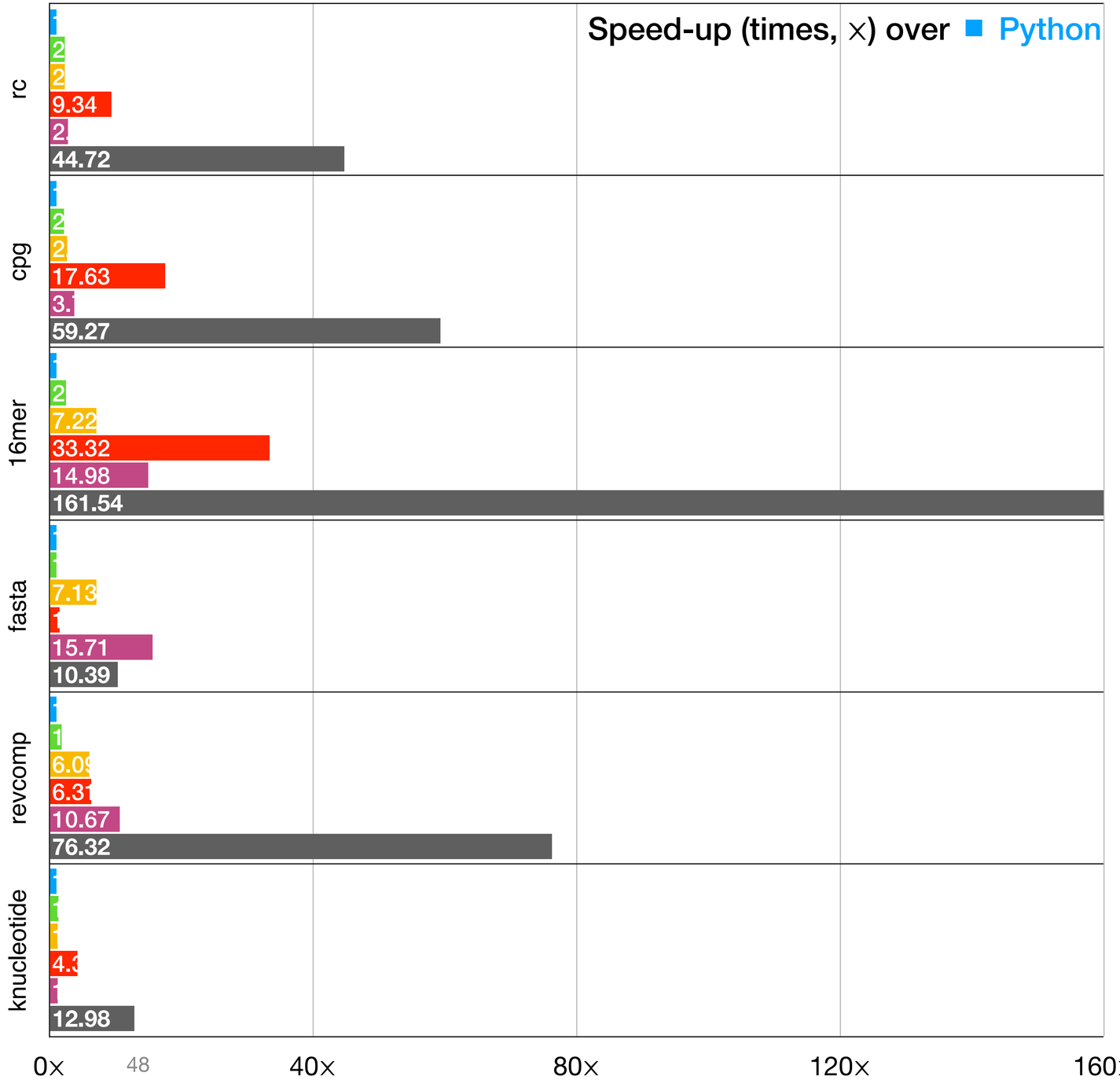
Benchmarks

- ■ **Python** is the reference implementation
- Compiled and JIT Pythons, such as ■ **Nuitka**, ■ **Shedskin** and ■ **PyPy** help a bit
- ■ **Julia** is similar…



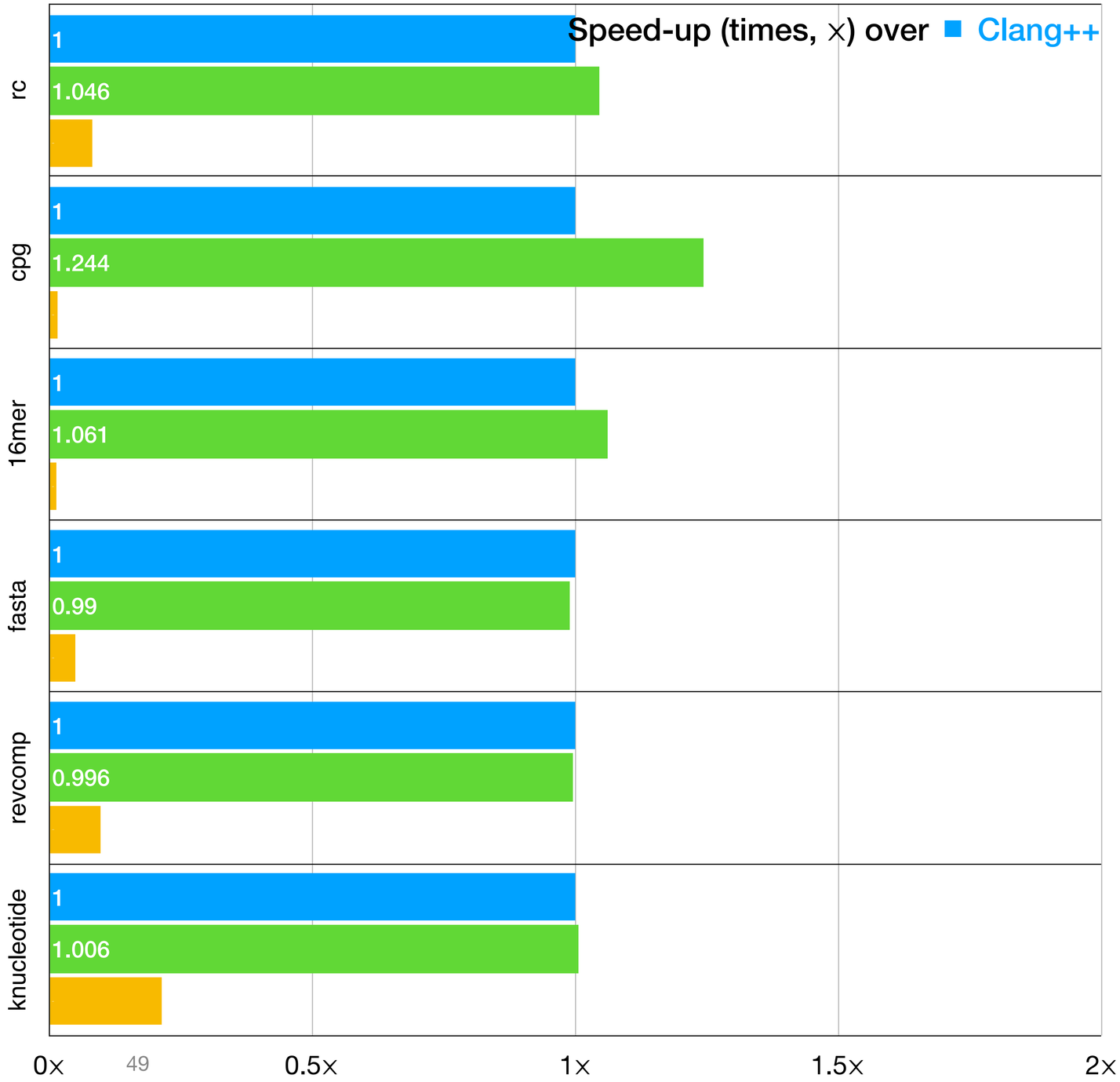
Benchmarks

- ■ **Python** is the reference implementation
- Compiled and JIT Pythons, such as
  - ■ **Nuitka**, ■ **Shedskin** and
  - ■ **PyPy** help a bit
- ■ **Julia** is similar…
- … but none come close to
  - ■ **Seq**
    - up to **160× speed-ups** over
      - ■ **Python**
    - or 1m 30s vs 4 hours and counting



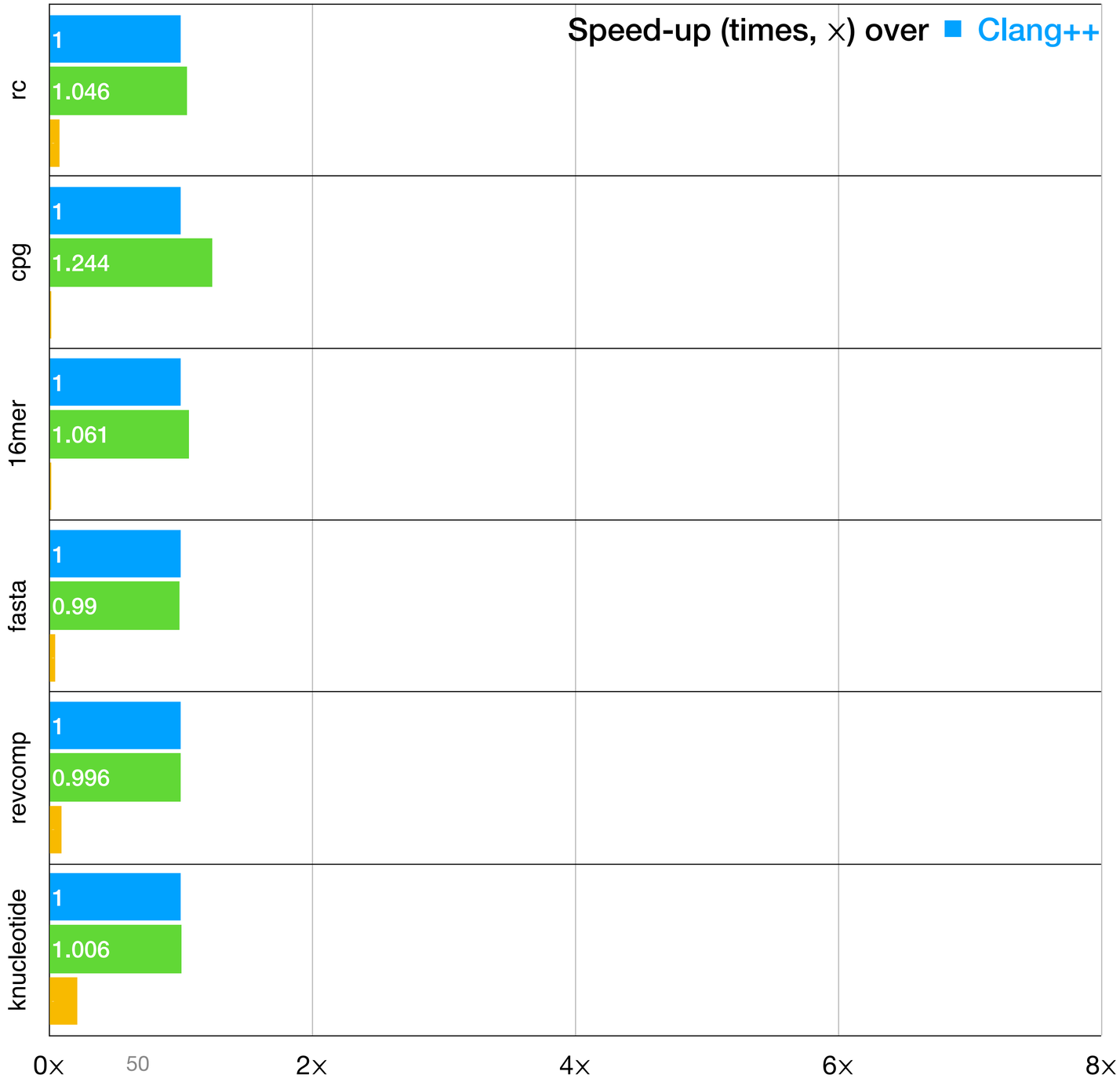
Benchmarks

- When compared to the reference
  - **Clang++** implementation…
- **g++** is similar to **Clang++**
- (this is **Python**)



Benchmarks

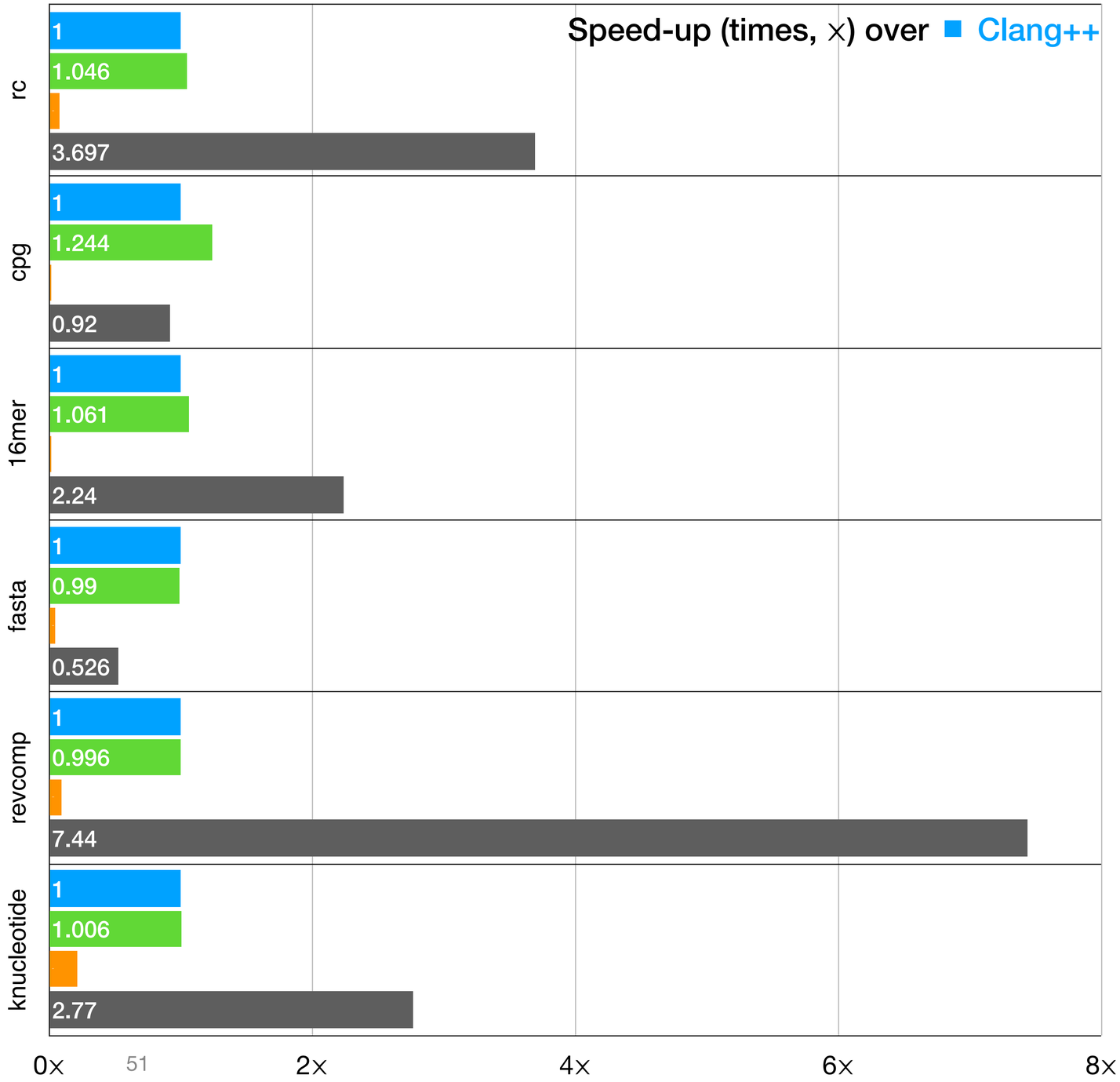
- When compared to the reference
  - **Clang++** implementation…
- **g++** is similar to **Clang++**
- (this is **Python**)



Benchmarks

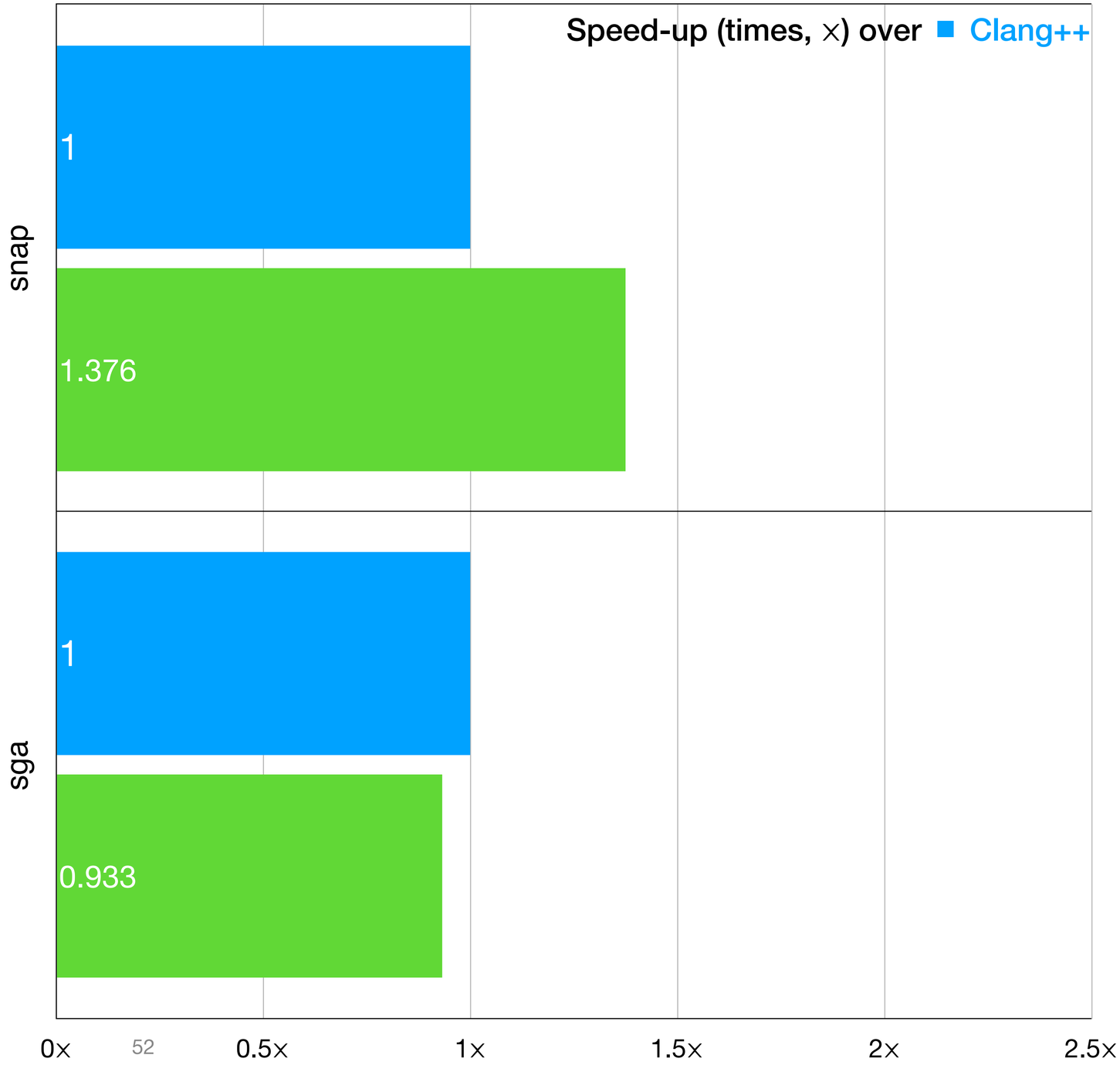
- When compared to the reference
  - **Clang++** implementation…
- **g++** is similar to **Clang++**
- (this is **Python**)
- and **Seq** outperforms

  even C++ in most experiments
  - up to 7× speed-ups over C++



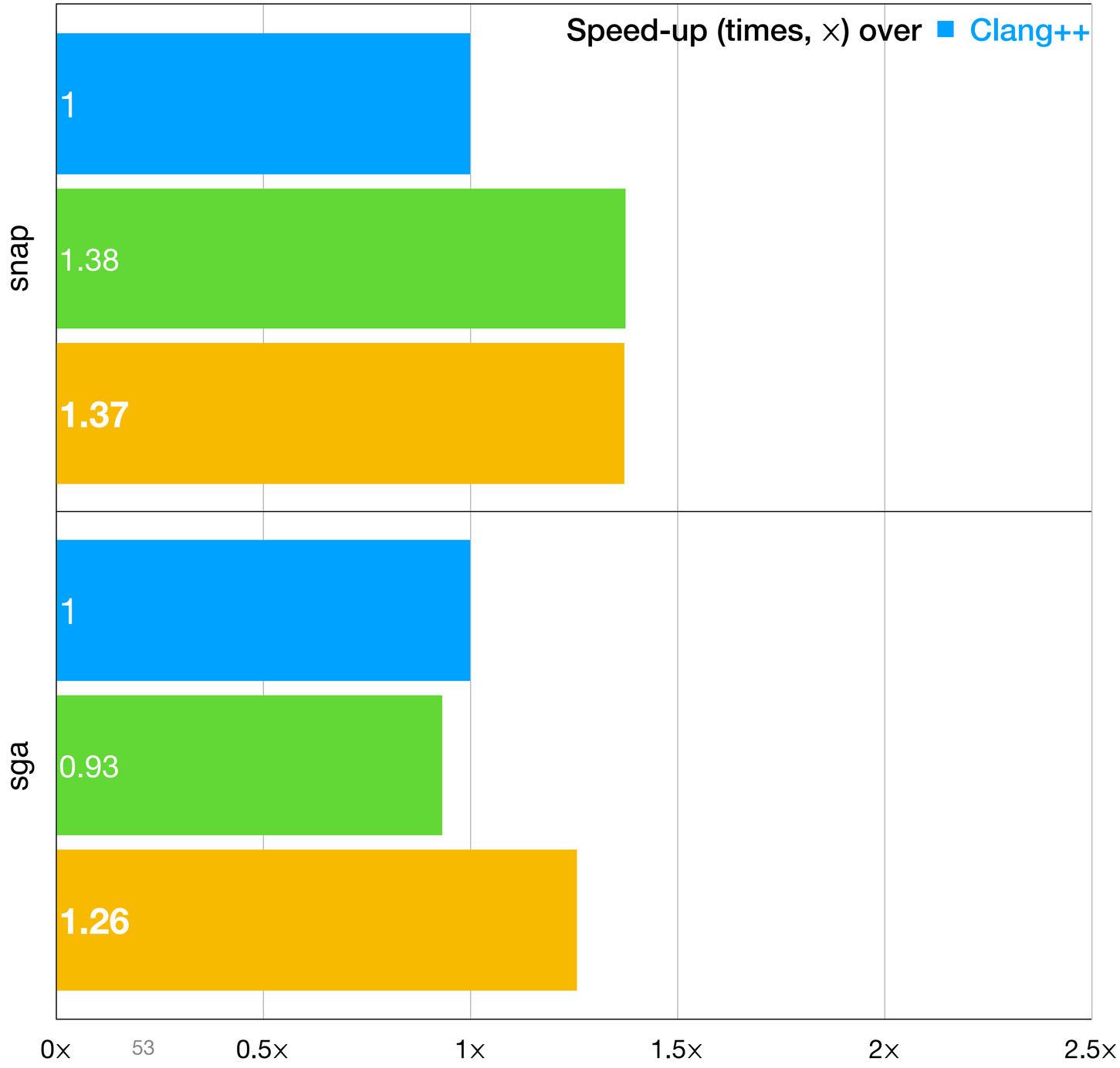
Benchmarks

- Finally, on genome index query benchmarks…
- reference ■ **Clang++** implementation and ■ **g++** are similar…



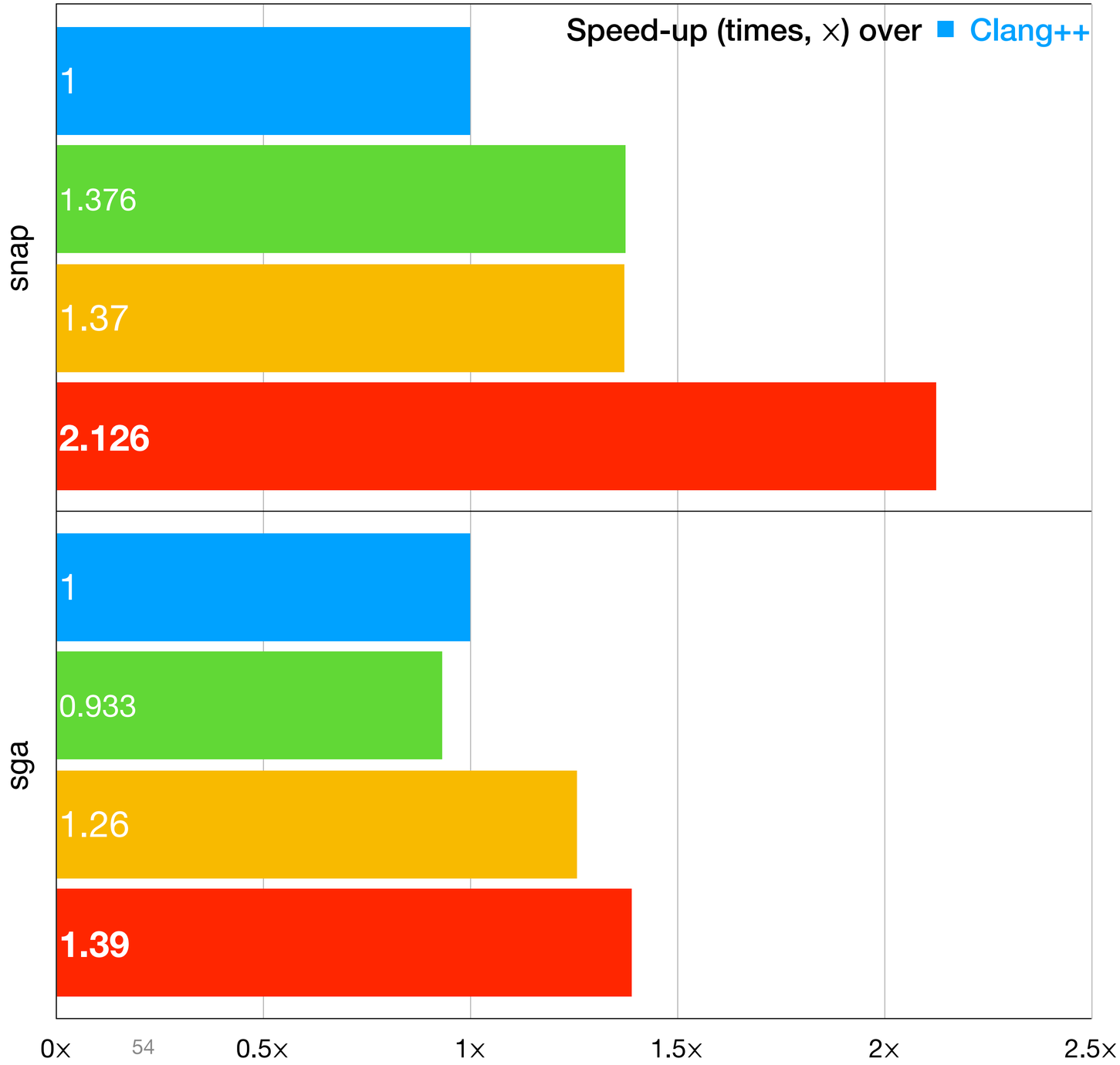
Benchmarks

- Finally, on genome index query benchmarks…
- reference ■ **Clang++** implementation and ■ **g++** are similar…
- ■ **Seq** can improve runtime up to 25%



Benchmarks

- Finally, on genome index query benchmarks…
- reference ■ **Clang++** implementation and ■ **g++** are similar…
- ■ **Seq** can improve runtime up to 25%
- and ■ **Seq with prefetch** boosts the performance of ■ **Seq** up to 50%!



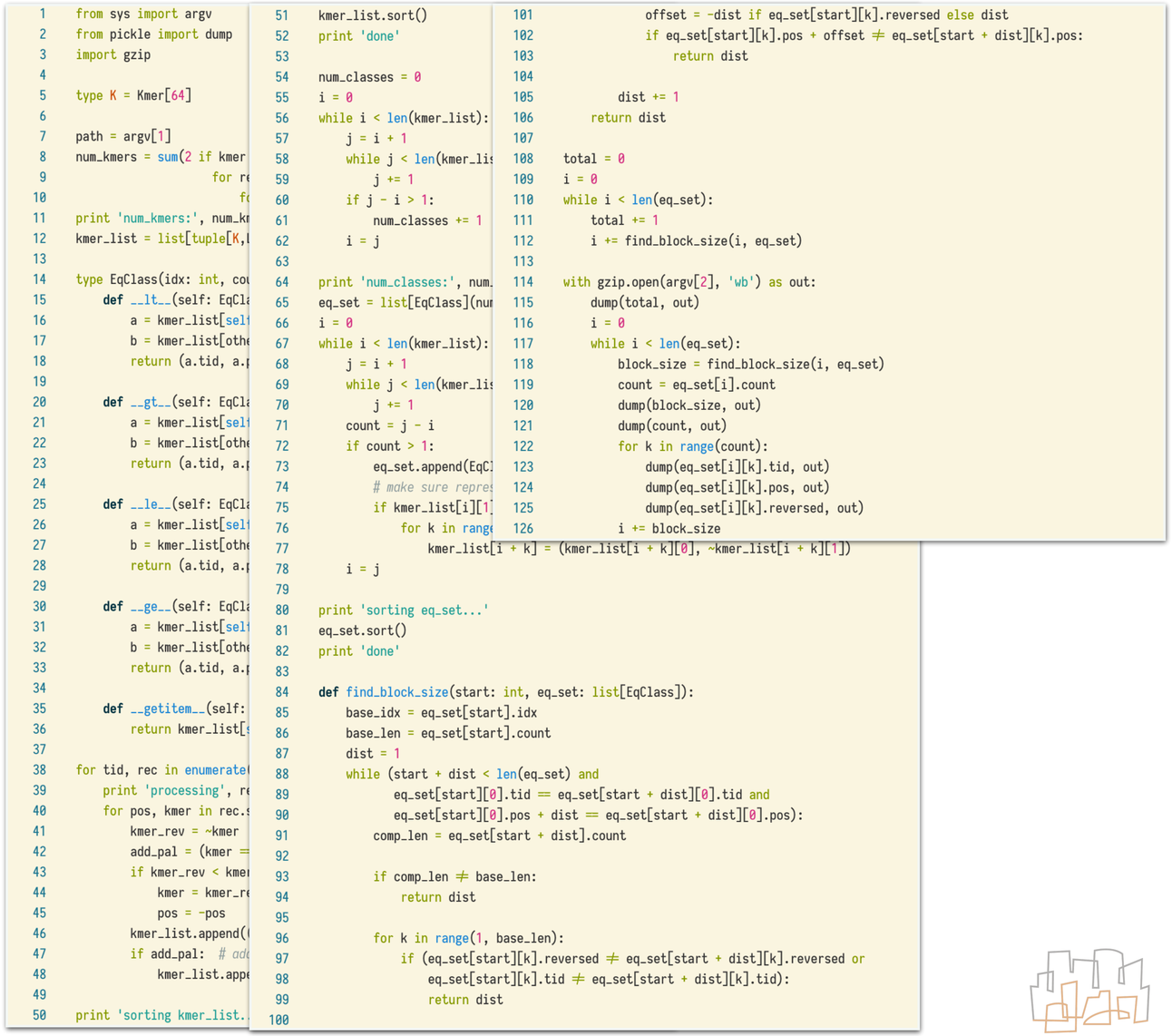
Seq: comparison with C++

- Re-implementation of homology table generator from CORA mapping software
- Ran on the whole human genome
- Highly optimized **C++**:
  LOC: 1,346 (27 screens)
  Runtime: 3+ hrs

MIT CSAIL

Seq: comparison with C++

- Re-implementation of homology table generator from CORA mapping software
- Ran on the whole human genome
- Highly optimized **C++**:
  LOC: 1,346 (27 screens)
  Runtime: 3+ hrs
- **Seq**:
  LOC: **126** (2½ screens— > 10× smaller)
  Runtime: **~30 minutes** (> 6× faster)



```
from sys import argv
from pickle import dump
import gzip

type K = Kmer[64]

path = argv[1]
num_kmers = sum(2 if kmer
                  for re
                      fo
print 'num_kmers:', num_km
kmer_list = list[tuple[K,I

type EqClass(idx: int, cou
    def __lt__(self: EqCla
        a = kmer_list[self
        b = kmer_list[othe
        return (a.tid, a.p

    def __gt__(self: EqCla
        a = kmer_list[self
        b = kmer_list[othe
        return (a.tid, a.p

    def __le__(self: EqCla
        a = kmer_list[self
        b = kmer_list[othe
        return (a.tid, a.p

    def __ge__(self: EqCla
        a = kmer_list[self
        b = kmer_list[othe
        return (a.tid, a.p

    def __getitem__(self:
        return kmer_list[s

for tid, rec in enumerate(
    print 'processing', re
    for pos, kmer in rec.s
        kmer_rev = ~kmer
        add_pal = (kmer ==
        if kmer_rev < kmer
            kmer = kmer_re
            pos = -pos
        kmer_list.append((
        if add_pal:  # add
            kmer_list.appe

print 'sorting kmer_list..
kmer_list.sort()
print 'done'

num_classes = 0
i = 0
while i < len(kmer_list):
    j = i + 1
    while j < len(kmer_lis
        j += 1
    if j - i > 1:
        num_classes += 1
    i = j

print 'num_classes:', num_
eq_set = list[EqClass](nu
i = 0
while i < len(kmer_list):
    j = i + 1
    while j < len(kmer_lis
        j += 1
    count = j - i
    if count > 1:
        eq_set.append(EqCl
        # make sure repres
        if kmer_list[i][1]
            for k in range
                kmer_list[i + k] = (kmer_list[i + k][0], ~kmer_list[i + k][1])
    i = j

print 'sorting eq_set...'
eq_set.sort()
print 'done'

def find_block_size(start: int, eq_set: list[EqClass]):
    base_idx = eq_set[start].idx
    base_len = eq_set[start].count
    dist = 1
    while (start + dist < len(eq_set) and
           eq_set[start][0].tid == eq_set[start + dist][0].tid and
           eq_set[start][0].pos + dist == eq_set[start + dist][0].pos):
        comp_len = eq_set[start + dist].count

        if comp_len != base_len:
            return dist

        for k in range(1, base_len):
            if (eq_set[start][k].reversed != eq_set[start + dist][k].reversed or
                eq_set[start][k].tid != eq_set[start + dist][k].tid):
                return dist

            offset = -dist if eq_set[start][k].reversed else dist
            if eq_set[start][k].pos + offset != eq_set[start + dist][k].pos:
                return dist

        dist += 1
    return dist

total = 0
i = 0
while i < len(eq_set):
    total += 1
    i += find_block_size(i, eq_set)

with gzip.open(argv[2], 'wb') as out:
    dump(total, out)
    i = 0
    while i < len(eq_set):
        block_size = find_block_size(i, eq_set)
        count = eq_set[i].count
        dump(block_size, out)
        dump(count, out)
        for k in range(count):
            dump(eq_set[i][k].tid, out)
            dump(eq_set[i][k].pos, out)
            dump(eq_set[i][k].reversed, out)
        i += block_size
```

Seq: summary

- A Python-based language for computational genomics
  - Speed of C
  - Ease and expressiveness of Python
  - Compile-time type checking
  - Genomic-related optimizations
  - Natural pipeline syntax and many other enhancements

Acknowledgements

MIT:

- **Ariya Shajii**
- Bonnie Berger
- Riyadh Baghdadi
- Saman Amarasinghe
- Hyunghoon Cho
- Alexander Leighton
- Lorenzo Di Tucci
- William Leiserson
- Fredrik Kjolstad
- Tao Schardl

University of Victoria:

- Jordan Watson
- Jodie Weldon

Get Seq at
https://seq-lang.org

Check the development
at **Github**

Read the paper
at **ACM DL**

MIT Computation and Biology Group
MIT Commit Compiler Group
University of Victoria Lab 0xTCG

Thank you!

Questions, and (hopefully) some answers

Get Seq at
https://seq-lang.org

Related work

	Domain	Target	Compi-lation	Unknown types allowed?	Full Python?	CPython runtime?	Multi-threading?
CPython	General	Bytecode	Interpreted	✓	✓	✓	✗
Seq	Bio.	LLVM IR	AOT	✗	✗	✗	✓
Cython	General	C	AOT	✗	✓	✓	✓
PyPy	General	Bytecode	Interpreted	✓	✓	✓	✗
Numba	Sci.	LLVM IR	JIT	✓	✗	✓	✗
Nuitka	General	C++	AOT	✓	✓	✓	✗
Pythran	Sci.	C++	AOT	✓	✓	✗	✓
Pyston	General	LLVM IR	JIT	✓	✓	✓	✗
HOPE	Astro.	C++	JIT	✗	✗	✗	✗
Shed Skin	General	C++	AOT	✗	✗	✗	✓
Grumpy	General	Go	AOT	✗	✗	✗	✓

Seq is the only Python-alike language that consistently matches the performance of C/C++