Helicase: Vectorized parsing and bitpacking of genomic sequences

Igor Martayan, Loup Lobet, Camille Marchet, Charles Paperman

RECOMB-Arch 2026, May 24th, Thessaloniki

Context: FASTA and FASTQ formats

```fasta
>header1
GATTAATCAC
>a longer header <o.o>
TTAGCTGTGC
GANNNTAGAT
CTA
>header3 ...
```

```fastq
@header1
GATTAATCAC
+
whXa~~~k5h
@header2
TTAGCTGTGCGANNNTAGATCTA
+
@This+is+a+valid+score!
```

Context: FASTA and FASTQ formats

```fasta
>header1
GATTAATCAC
>a longer header <o.o>
TTAGCTGTGC
GANNNTAGAT
CTA
>header3 ...
```

```fastq
@header1
GATTAATCAC
+
whXa~~~k5h
@header2
TTAGCTGTGCGANNNTAGATCTA
+
@This+is+a+valid+score!
```

Context: FASTA and FASTQ formats

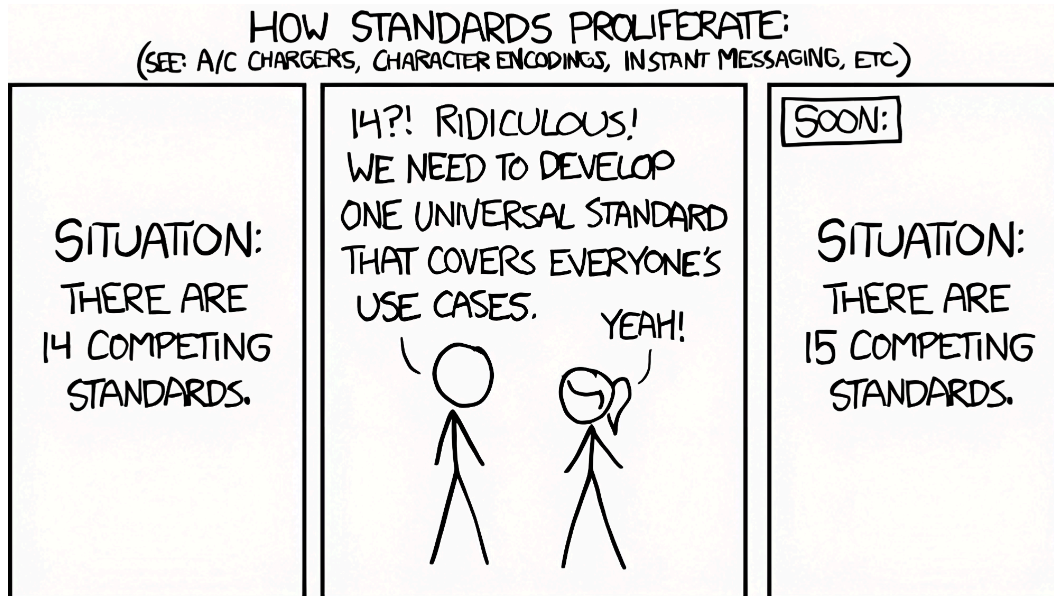
fasta

```
>header1
GATTAATCAC
>a longer header <o.o>
TTAGCTGTGC
GANNNTAGAT
CTA
>header3 ...
```

fastq

```
@header1
GATTAATCAC
+
whXa~~~k5h
@header2
TTAGCTGTGCGANNNTAGATCTA
+
@This+is+a+valid+score!
```

Why another FASTA/Q parser?



xkcd.com/927

Popular parsers:

`kseq`, `needletail`, `paraseq`...

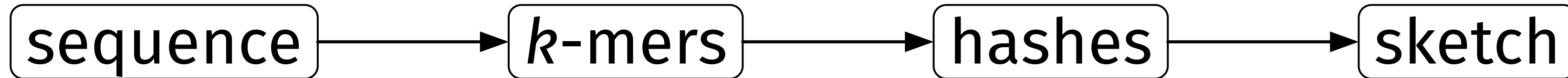
→ all rely on `memchr`, no advanced vectorization

→ especially slow on multiline FASTA

→ only output ASCII sequences, no bitpacking

From text-based formats to bitpacked streams

Many simple tasks can be formulated as streams, e.g.



We want to reduce the overhead of intermediate representations

→ take advantage of modern hardware for high throughput

→ bitpack data for better cache efficiency

Marking FASTA headers with carry propagation

Goal: mark everything between > and ↵ *branchlessly*

input	>	c	h	r	1	↵	C	G	G	A	C	↵	A	C	G	T	↵	>	c	>	h	↵	C
↵	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0
>	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0
!↵	1	1	1	1	1	0	1	1	1	1	1	0	1	1	1	1	0	1	1	1	1	0	1

Marking FASTA headers with carry propagation

Goal: mark everything between > and ↵ *branchlessly*

input	>	c	h	r	1	↵	C	G	G	A	C	↵	A	C	G	T	↵	>	c	>	h	↵	C
↵	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0
>	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0
!↵	1	1	1	1	1	0	1	1	1	1	1	0	1	1	1	1	0	1	1	1	1	0	1
> + !↵	0	0	0	0	0	1	1	1	1	1	1	0	1	1	1	1	0	0	0	1	0	1	1

Marking FASTA headers with carry propagation

Goal: mark everything between > and ↵ *branchlessly*

input	>	c	h	r	1	↵	C	G	G	A	C	↵	A	C	G	T	↵	>	c	>	h	↵	C
↵	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0
>	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0
!↵	1	1	1	1	1	0	1	1	1	1	1	0	1	1	1	1	0	1	1	1	1	0	1
> + !↵	0	0	0	0	0	1	1	1	1	1	1	0	1	1	1	1	0	0	0	1	0	1	1
⊕ !↵	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	1	1	0	1	1	0

Marking FASTA headers with carry propagation

Goal: mark everything between > and ↵ *branchlessly*

input	>	c	h	r	1	↵	C	G	G	A	C	↵	A	C	G	T	↵	>	c	>	h	↵	C
↵	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0
>	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0
!↵	1	1	1	1	1	0	1	1	1	1	1	0	1	1	1	1	0	1	1	1	1	0	1
> + !↵	0	0	0	0	0	1	1	1	1	1	1	0	1	1	1	1	0	0	0	1	0	1	1
⊕ !↵	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	1	1	0	1	1	0
∨ >	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	0

→ we produce a stream of annotation bits

Bitpacked representations of DNA

A	C	T	G
01000001	01000011	01010100	01000111

Packed and columnar representations:

sequence	G	A	T	T	A	C	A	T
packed	11	00	10	10	00	01	00	10
col hi	1	0	1	1	0	0	0	1
col lo	1	0	0	0	0	1	0	0

Bitpacked representations of DNA

A	C	T	G
01000001	01000011	01010100	01000111

Packed and columnar representations:

sequence	G	A	T	T	A	C	A	T
packed	11	00	10	10	00	01	00	10
col hi	1	0	1	1	0	0	0	1
col lo	1	0	0	0	0	1	0	0

What about non-ACTG characters?

Detecting non-ACTG characters

```
1 LUT ← ['A', _, 'C', _, 'T', _, 'G', _]        ▷ constant lookup table mapping bits to ACTG
2 v_bits ← v_data & 0b110                        ▷ lossy 2-bit encoding
3 v_lookup ← Lookup* (LUT, v_bits)               ▷ map bits to ACTG using the lookup table
4 v_upper ← v_data & !0b00100000                 ▷ uppercase the input
5 v_eq ← CmpEq (v_lookup, v_upper)
6 return MoveMask** (v_eq)                       ▷ 1s indicating ACTG
```

* implemented with a shuffle

** on ARM: implemented with interleaving and byte-wise packing

Handling non-ACTGs

2 different strategies available:

```
G  A  T  T  N  N  A  C  A
```

$\rightarrow$ *split* sequences into chunks of ACTGs

```
[11 00 10 10]     [00 01 00]
```

$\rightarrow$ *lossy encoding* of non-ACTGs + *bitmask* marking them

```
[11 00 10 10 11 11 00 01 00]
[ 0  0  0  0  1  1  0  0  0]
```

Specifying what we want to parse at compile time

```
1 const CONFIG: Config = ParserOptions::default()
2       .ignore_headers()
3       .dna_columnar()
4       .split_non_actg()
5       .config();
```

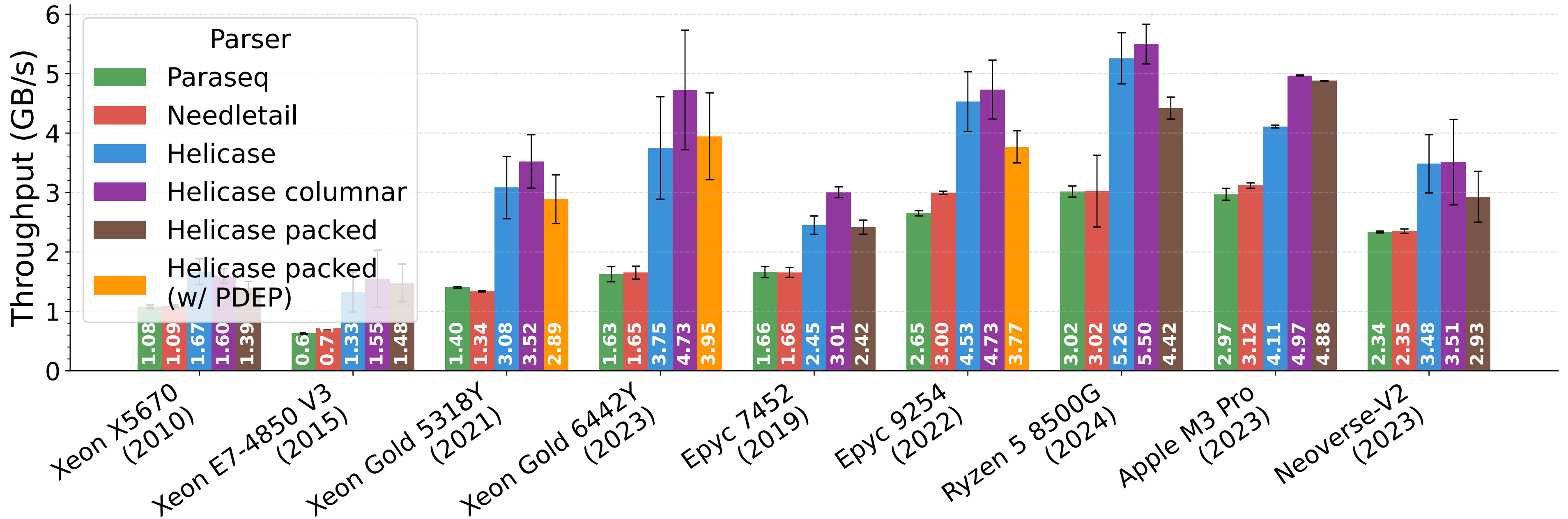
Unused fields / representations are eliminated by the compiler

A Rust API similar to needletail

```
1 let mut parser =
  FastxParser::<CONFIG>::from_file(&path)?;
2 while let Some(_) = parser.next() {
3   let header = parser.get_header();
4   let seq = parser.get_dna_columnar();
5   // ...
6 }
```

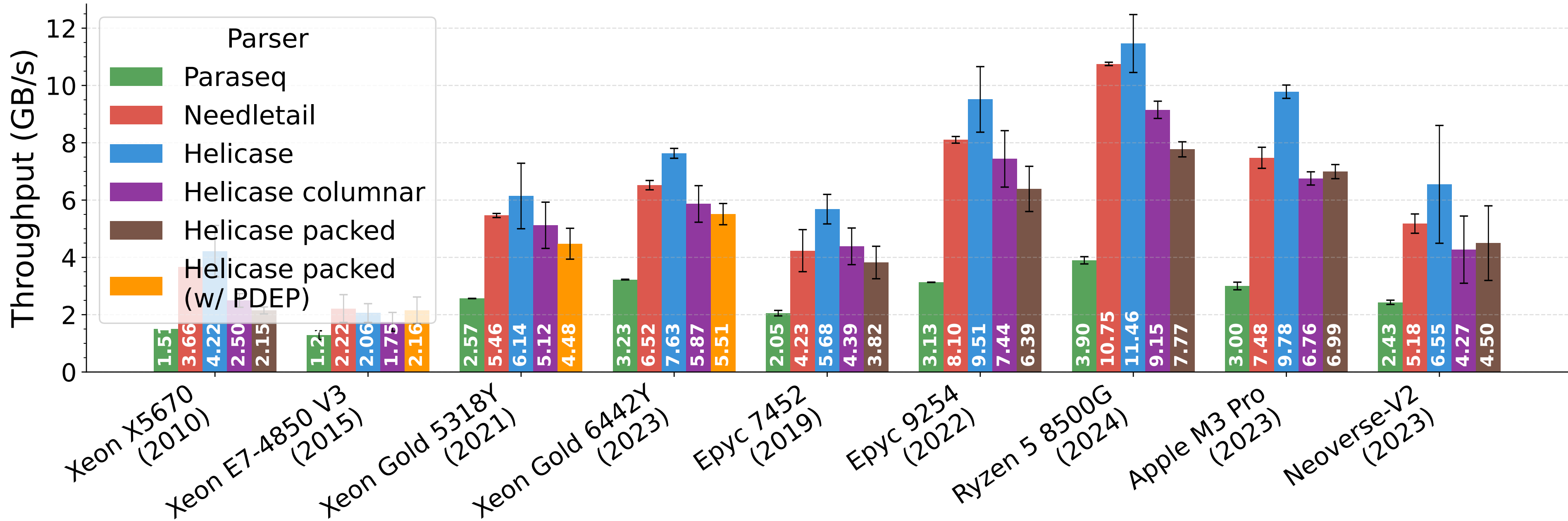
→ automatic format detection, transparent decompression

Benchmarks: parsing multiline FASTA from disk



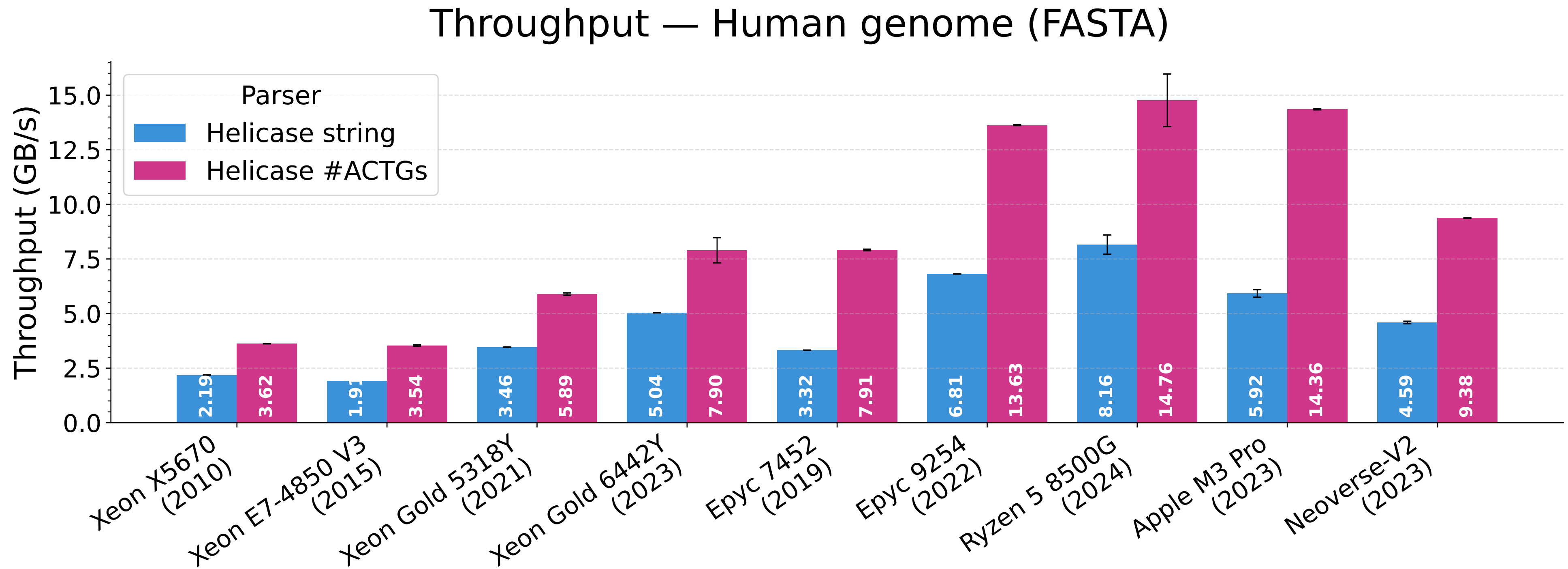
$\rightarrow$ 50-100% faster, columnar is the fastest

Benchmarks: parsing short reads FASTQ <u>from disk</u>



→ 10-20% faster, bitpacking 20-30% more expensive

Benchmarks: counting ACTG bases in RAM



→ specialization can speed up compute-limited tasks

What's next?

Helicase improvements:

- API for parallelization (without producer/consumer)
- built-in k-mer iterators
- Python/C/C++ bindings

github.com/imartayan/helicase

Avoiding (de)compression bottlenecks:

- binseq (Teyssier et al., 2025)
- mim (Patro et al., 2025)

What’s next?

Helicase improvements:

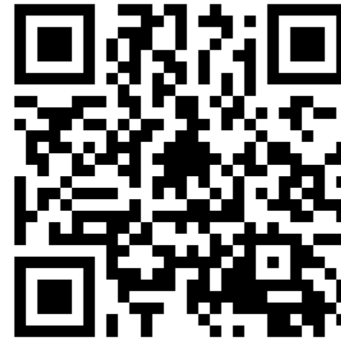
- API for parallelization (without producer/consumer)
- built-in k-mer iterators
- Python/C/C++ bindings

github.com/imartayan/helicase

Thank you!

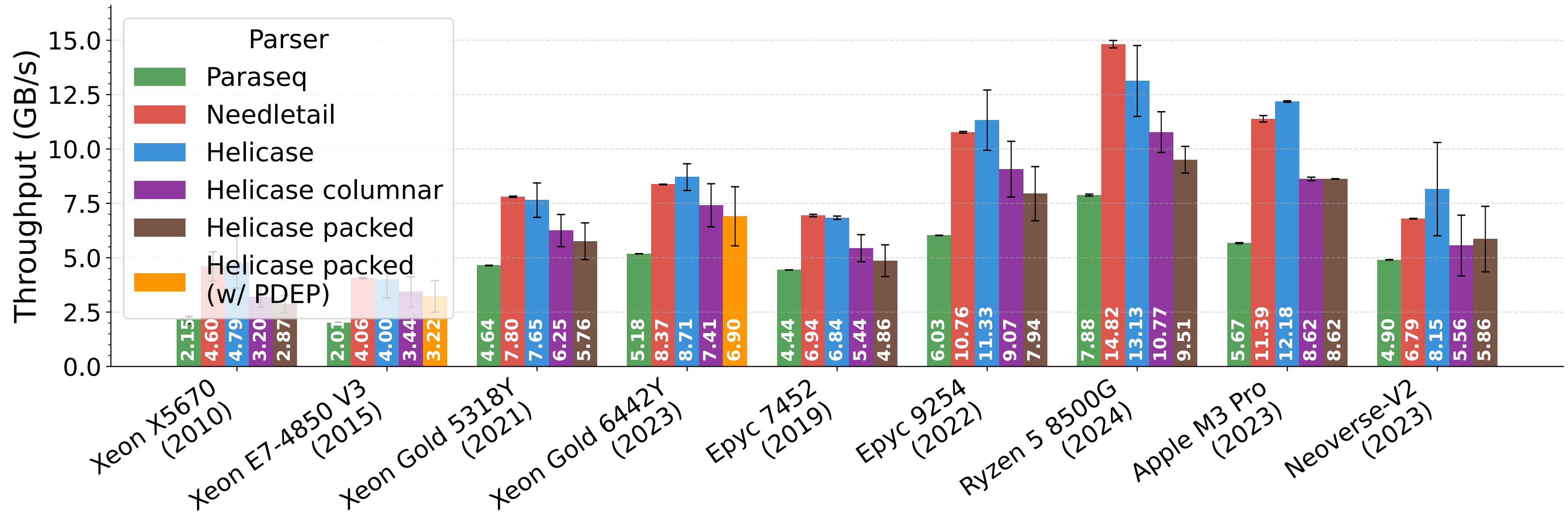
Avoiding (de)compression bottlenecks:

- binseq (Teyssier et al., 2025)
- mim (Patro et al., 2025)

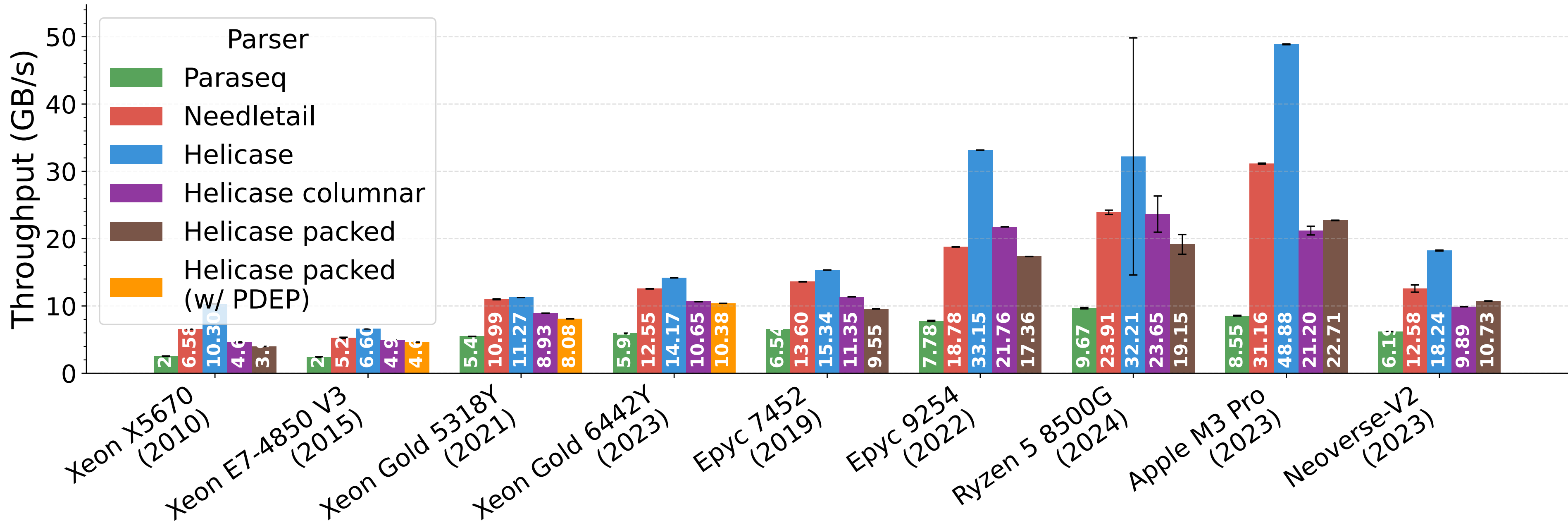


Appendix

Benchmarks: parsing long reads FASTQ from disk

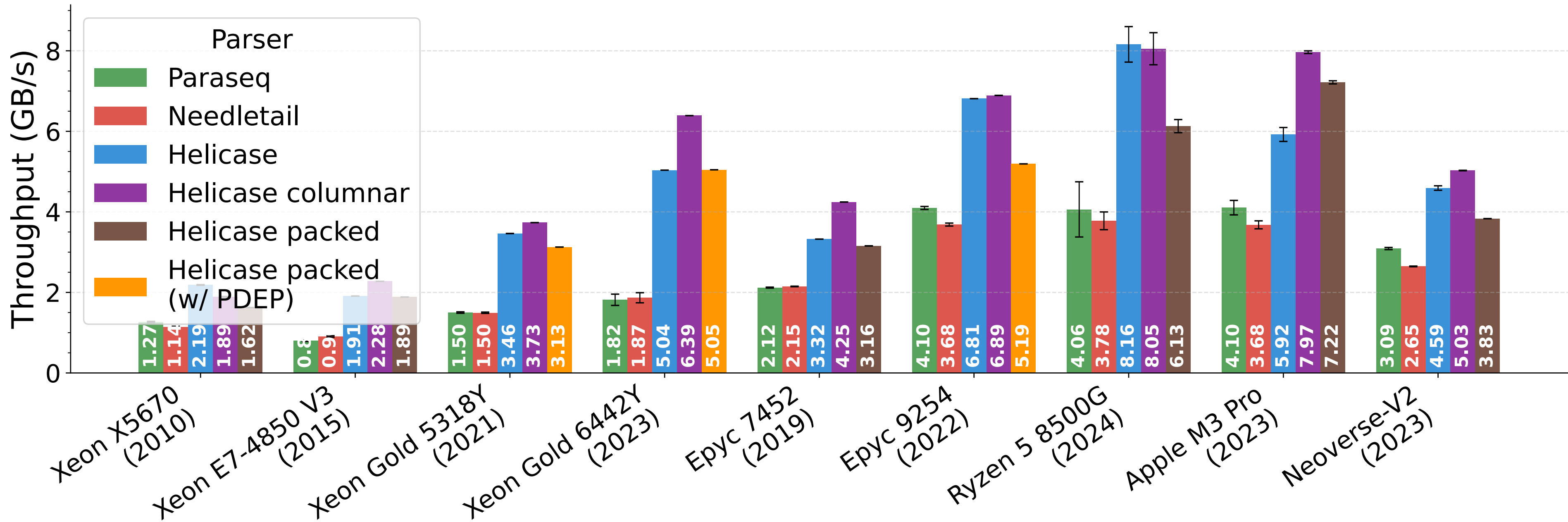


Benchmarks: parsing long reads FASTQ <u>in RAM</u>



→ zero-copy, bandwidth-limited!

Benchmarks: parsing multiline FASTA <u>in RAM</u>



→ slower throughput than FASTQ, compute-limited