who owns the bytes?

making numpy-ts as fast as native

nicolas dupont · jsnation us 2026

python's whole data stack sits on numpy

javascript has nothing quite like it

stdlib-js · ndarray · tf.js

each covers part of this, with a different api or an ml focus



numpy-ts

same api. same behavior. in typescript.

node | deno | bun | edge runtimes | browsers

python - numpy

```python
x = np.random.standard_normal((1024, 64))
x = (x - x.mean(axis=0)) / x.std(axis=0)
u, s, vt = np.linalg.svd(x, full_matrices=False)
spectrum = np.abs(np.fft.rfft(x[:, 0]))
```

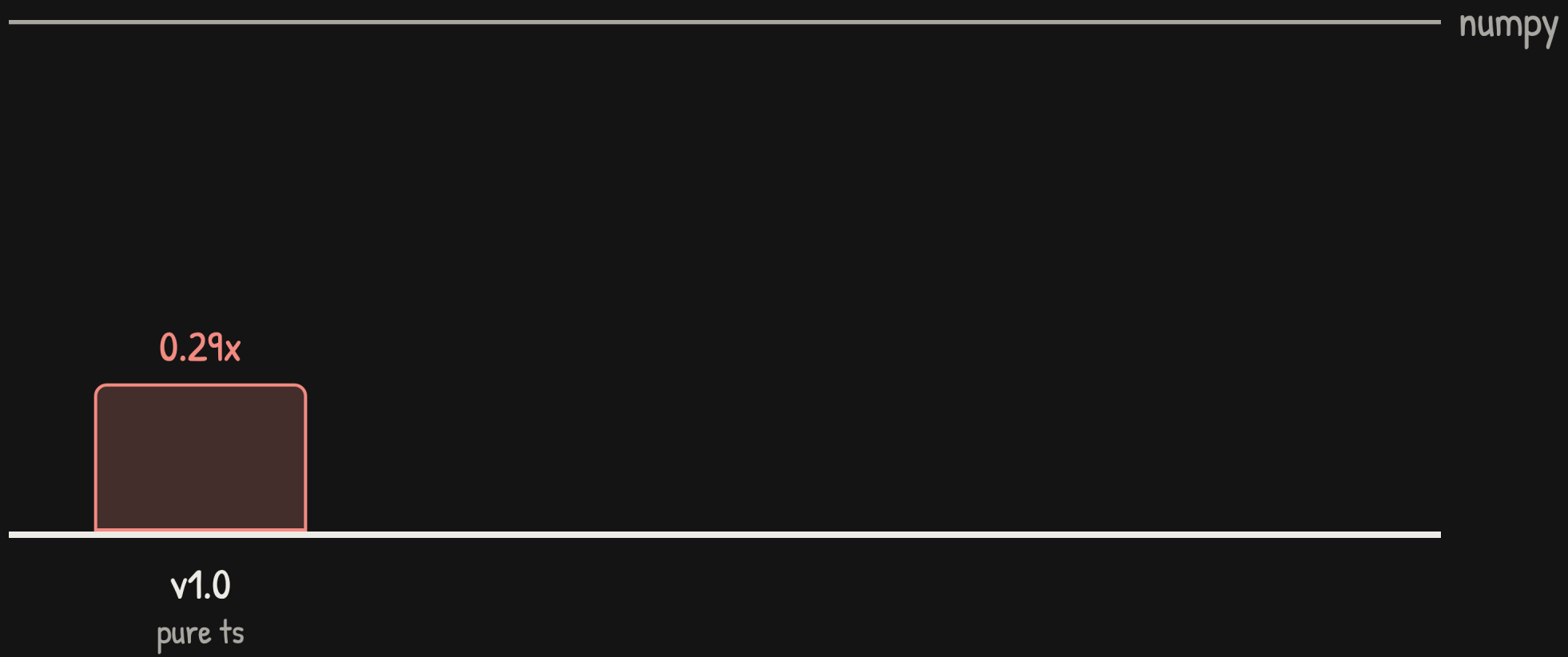
typescript - numpy-ts

```typescript
let x = np.random.standard_normal([1024, 64]);
x = x.subtract(x.mean(0)).divide(x.std(0));
const { u, s, vt } = np.linalg.svd(x, false);
const spectrum = np.abs(np.fft.rfft(x.slice(':', '0')));
```

getting the api right was the easy part.

then i ran the benchmarks.

arithmetic ops · geometric mean vs numpy



>100x slower

than numpy, on some ops

the jit isn't the problem

```
function add(a: Float64Array, b: Float64Array,
             out: Float64Array) {
  for (let i = 0; i < a.length; i++) {
    out[i] = a[i] + b[i];
  }
}
```

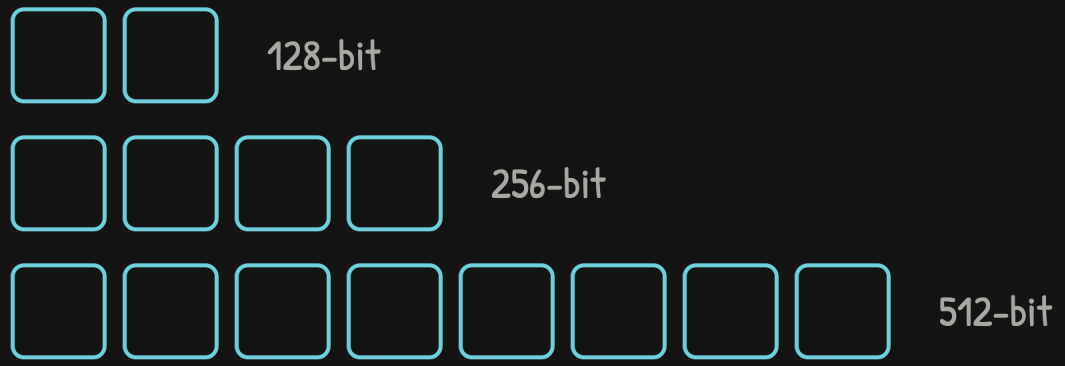
monomorphic, typed: compiles to roughly scalar c

scalar c isn't the bar

scalar · 1 lane



simd · float64 lanes per instruction



behind numpy: **blas** and **pocketfft**

"obviously the answer is wasm."

one big numpy.wasm?

too big

not tree-shakeable

async loading

pyodide

real numpy, compiled to wasm

big download

~8 mb before the first array

async
startup

python ↔ js
marshalling

zig microkernels



one module per kernel · sync init · only what you import ships (numpy-ts/core)

why zig

small wasm, no runtime

explicit simd vectors

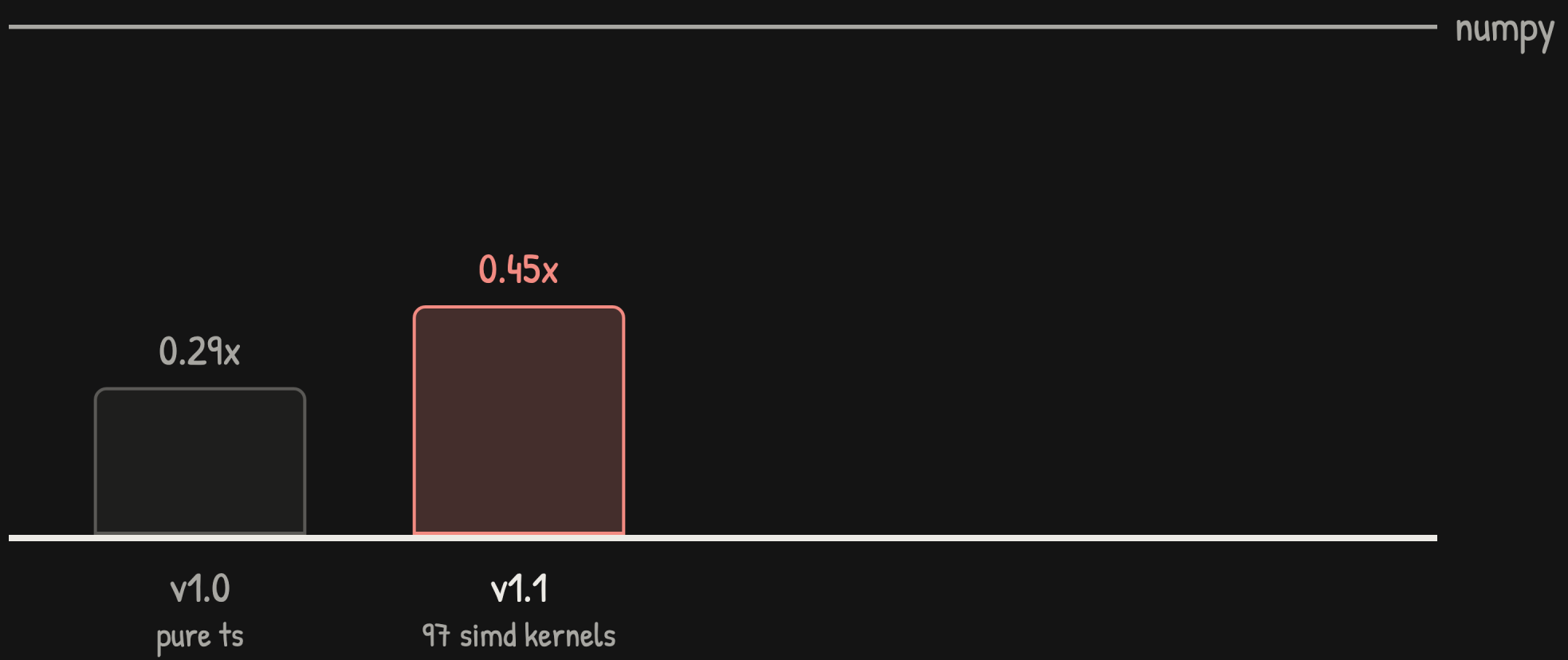
comptime: every dtype from one kernel

why not webgpu

not in every runtime

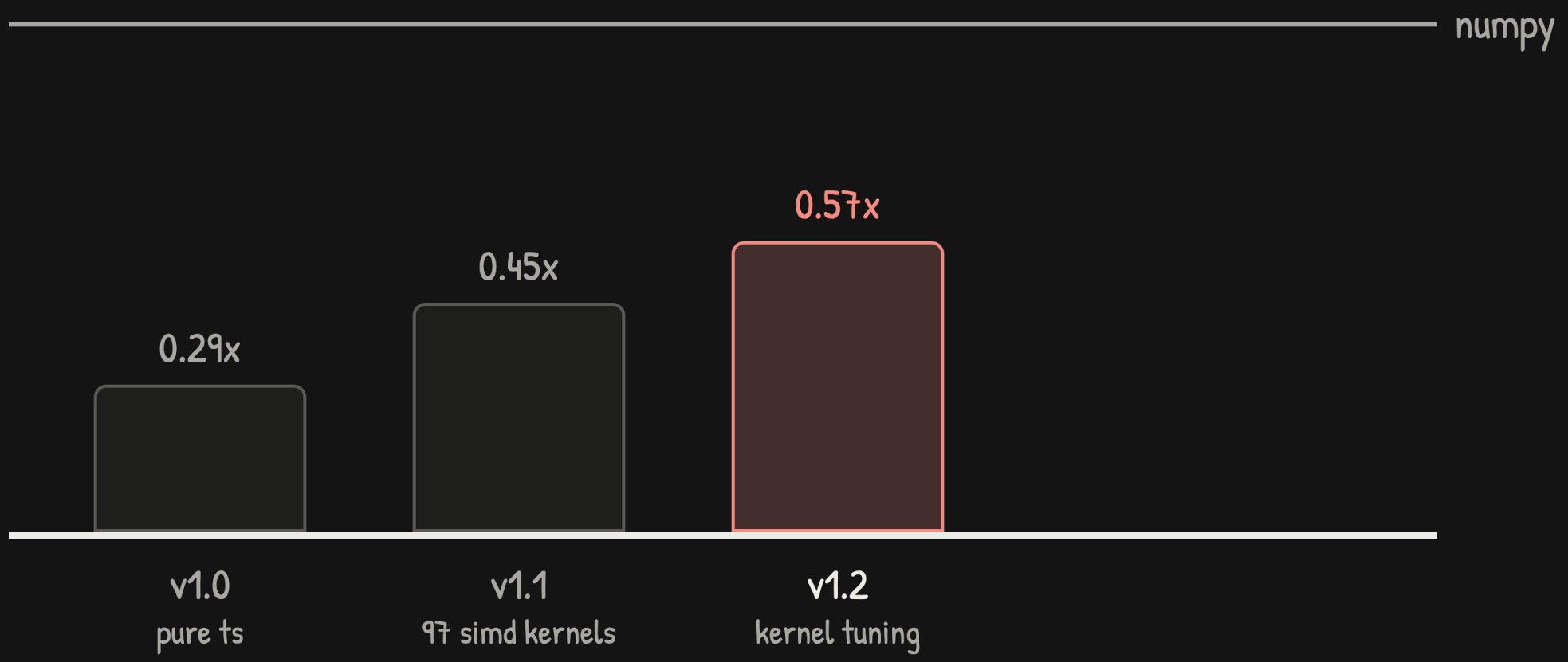
dispatch + readback costs more than the op itself on small and medium arrays

arithmetic ops · geometric mean vs numpy



so i optimized it.

arithmetic ops · geometric mean vs numpy



after two releases of kernel work, arithmetic was still

~2x slower than numpy

where is the bottleneck?

~~js ↔ wasm calls~~

<10 nanoseconds in v8 and spidermonkey

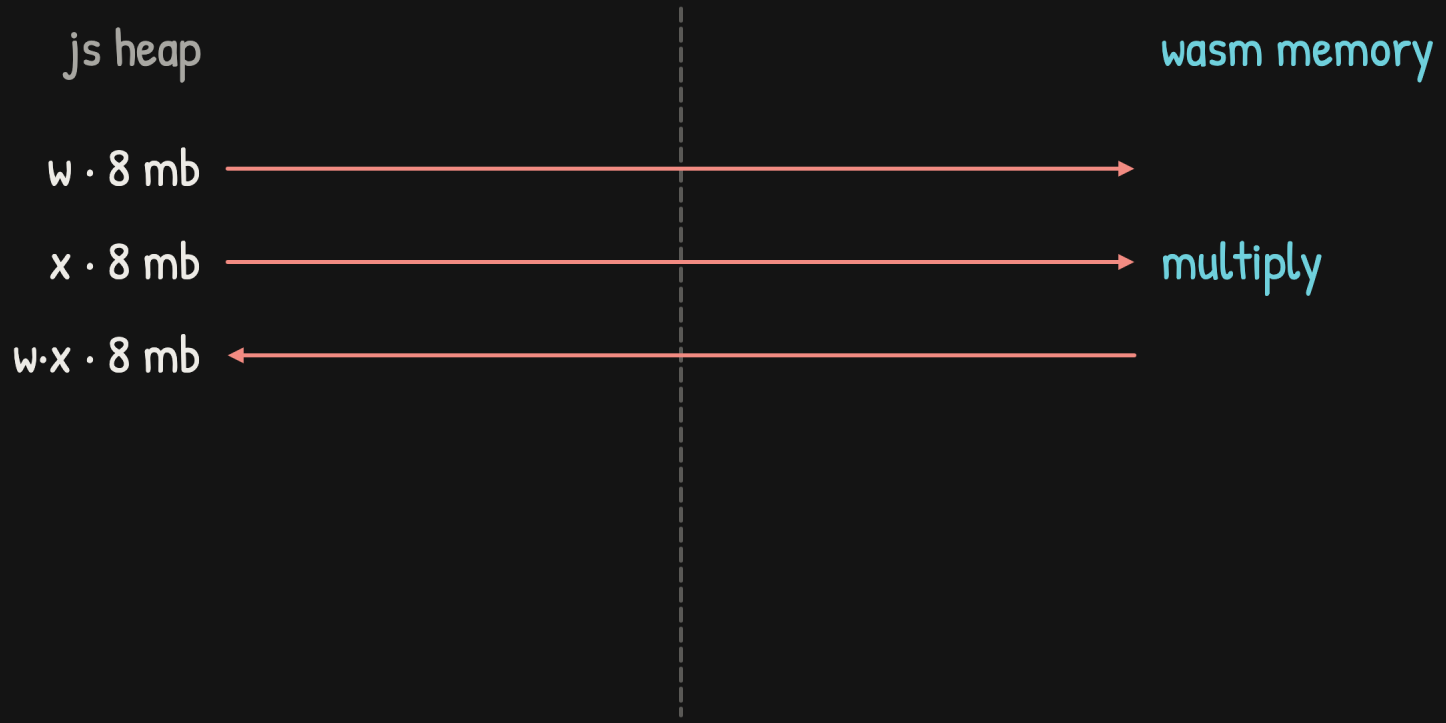
```
const y = w.multiply(x).add(b);
```

three float64 arrays · 1m elements · 8 mb each

```
w.multiply(x).add(b)
```

24 mb

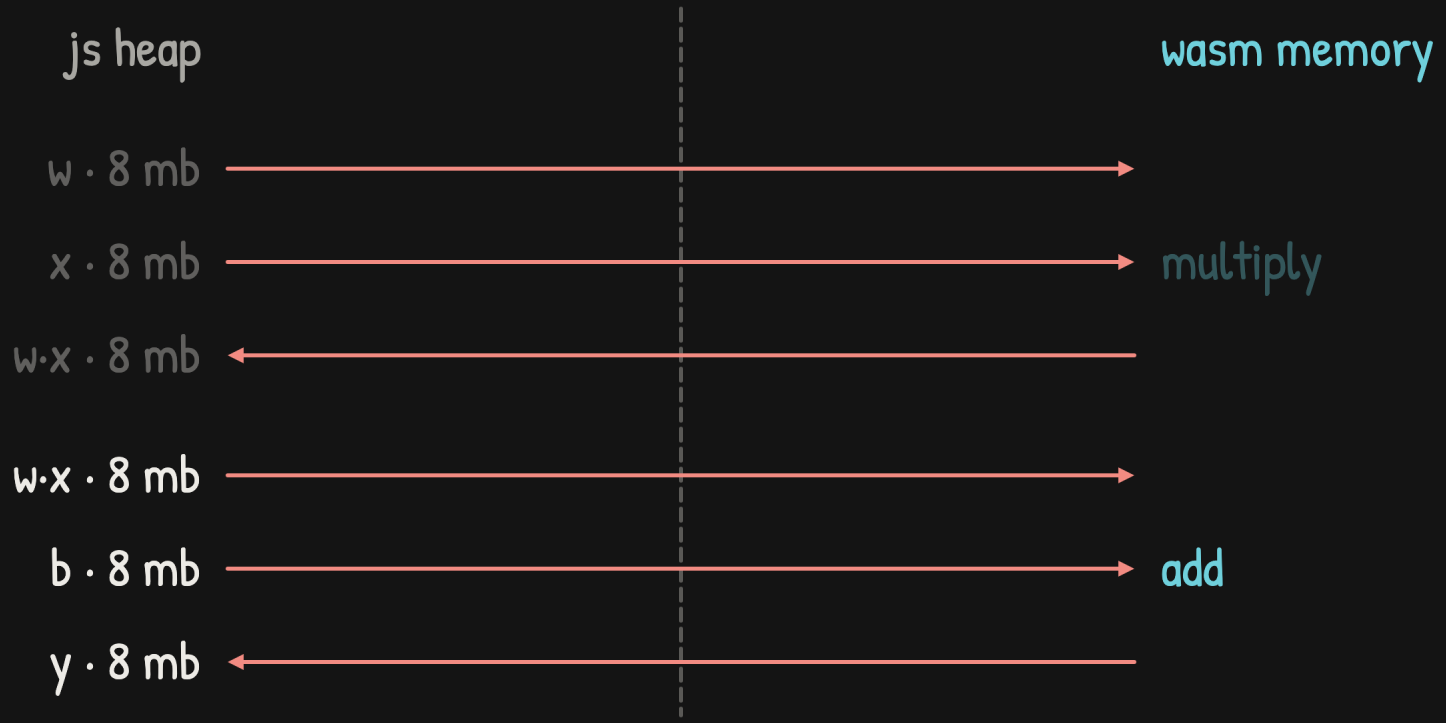
across the boundary



```
w.multiply(x).add(b)
```

48 mb

to produce 8 mb



add reads each element once and writes it once

so does memcpy

one v1.2 add call · 1m float64

49% of it is memcpy



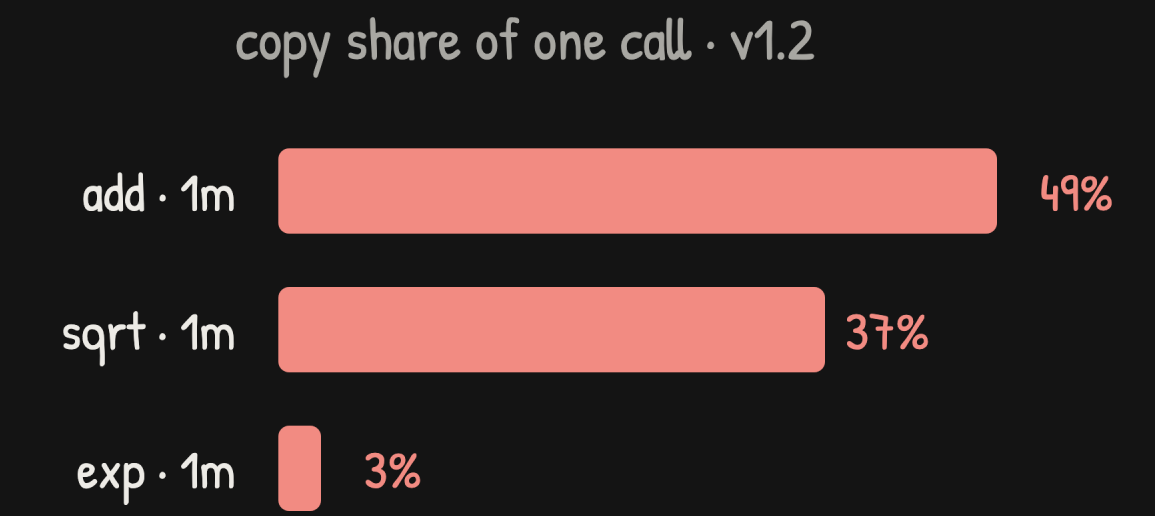
kernel, bookkeeping, js result · 295 µs

3 copies · 288 µs

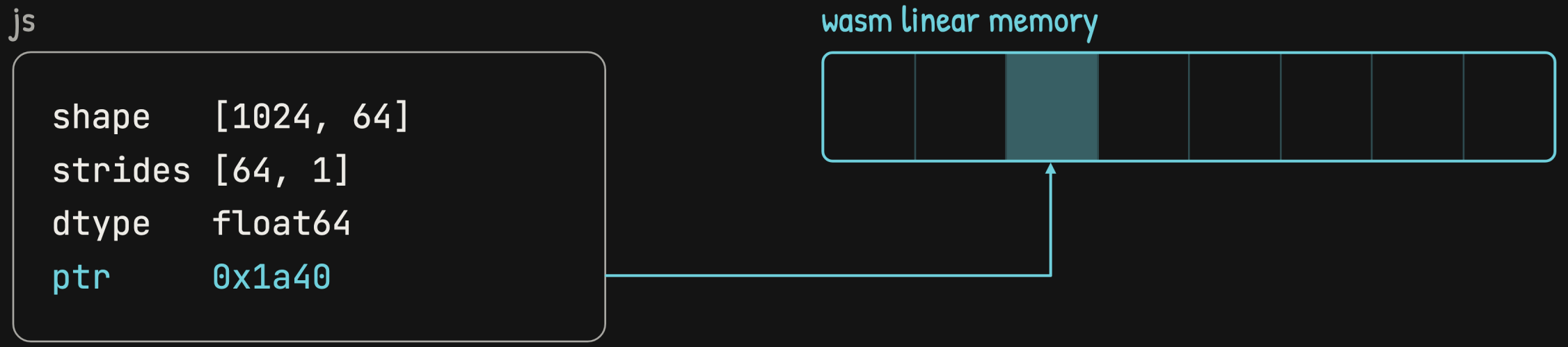
583 µs total

copies and kernel both scale with n

a faster kernel only makes the copies a bigger share



cheap ops pay ~30–50% · transcendentals pay ~1–10%



metadata stays in js. bytes live in wasm.

kernels get pointers.

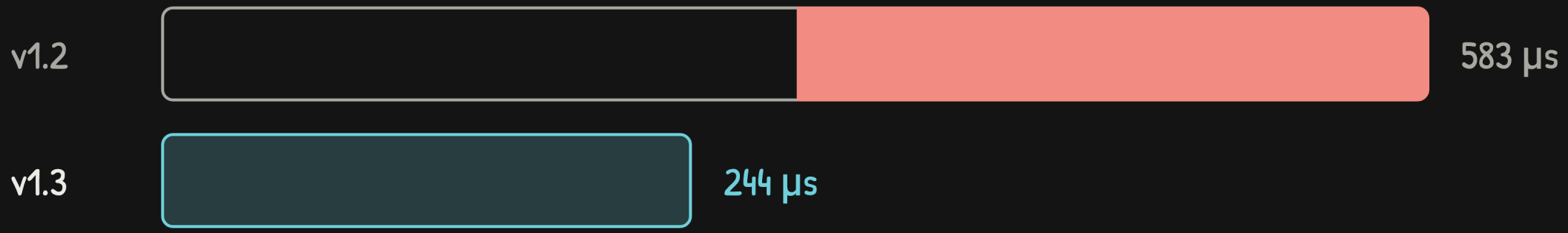
`w.multiply(x).add(b)` · 1m

1.2 ms → 0.5 ms

v1.2 → v1.3

one add call · 1m float64

339 µs saved



the 288 µs of copies, plus most of the js result allocation

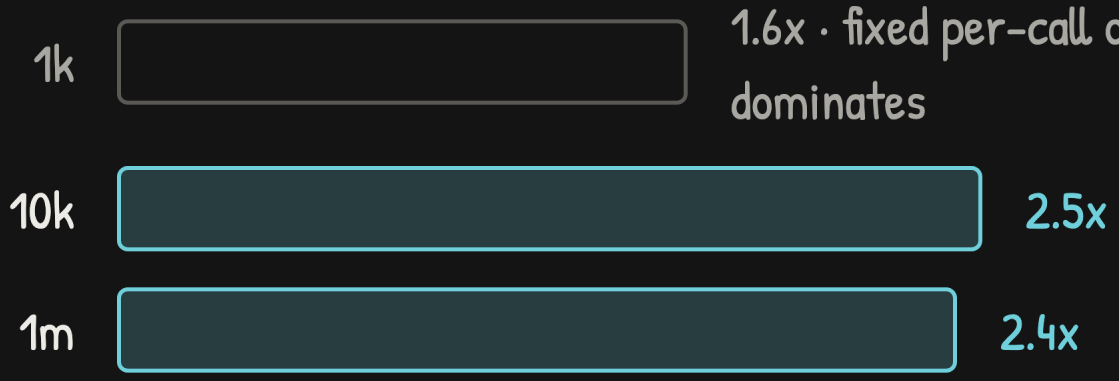
add and multiply · 1m · v1.2 → v1.3

2.4x faster

same kernels: add.zig only gained tests

removing copies scales

speedup from v1.2 to v1.3



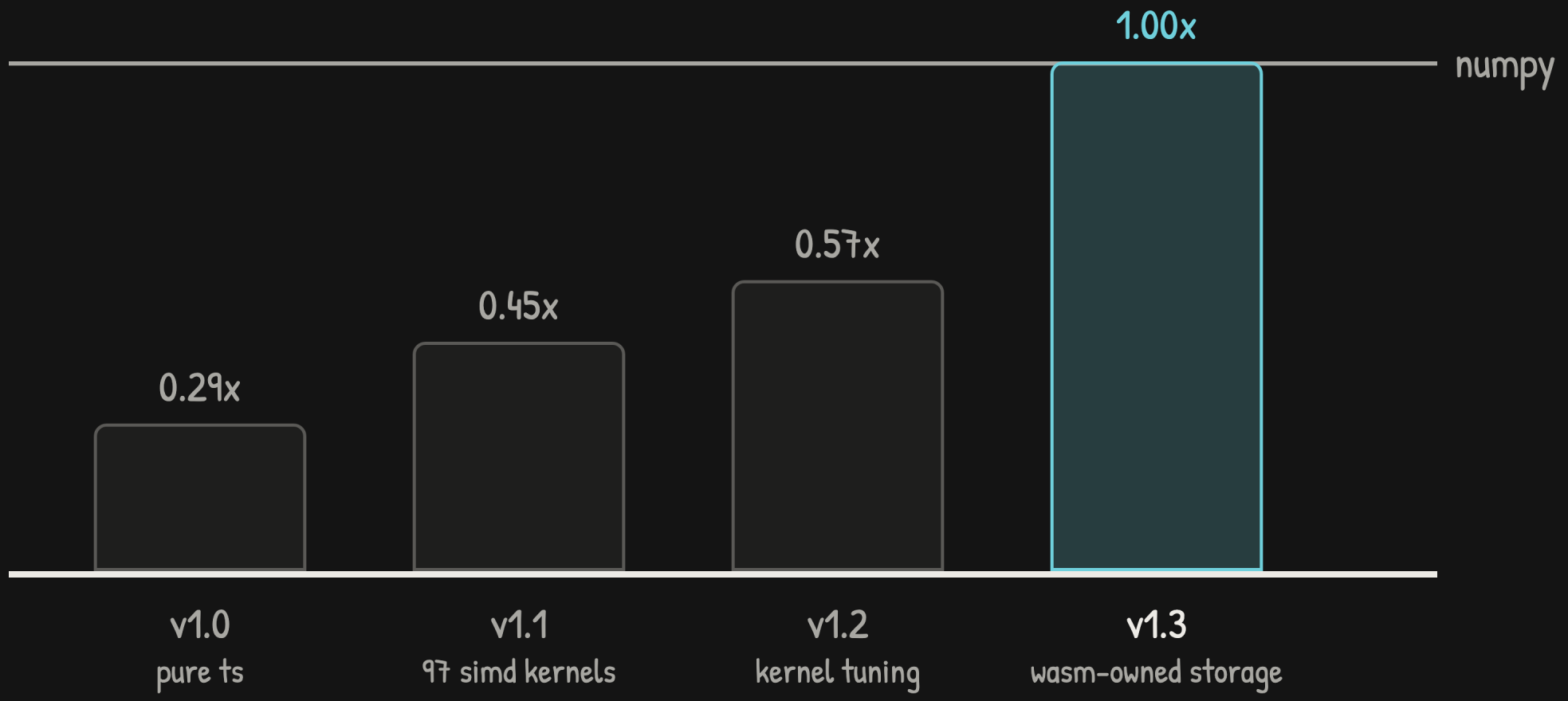
add · same kernel in both versions

add · 1m vs numpy · single benchmark

0.87x → 2.1x

thread for thread: numpy's elementwise ops run on one thread too

arithmetic ops · geometric mean vs numpy



same benchmarks in every version

the twist

numpy's add kernel is faster than ours

numpy · out=	201 µs
numpy-ts · pooled	244 µs

the twist

numpy's add kernel is faster than ours

numpy · out=	201 µs
numpy-ts · pooled	244 µs
numpy · default call (fresh output)	506 µs

performance follows who owns the bytes

the downside

owning the bytes means running your own heap

```
const view = new Float64Array(mem.buffer);
mem.grow(1);
view.length; // 0: detached
```

so the heap is sized once: initial === maximum

what about big arrays?

256 mib
default heap

`configureWasm()`
before the first op

pool full
falls back to js arrays

4 gb
wasm32's ceiling

coming soon

`toResizableBuffer()`

views that survive growth

`memory64`

past the 4 gb ceiling

not in every runtime numpy-ts targets, yet

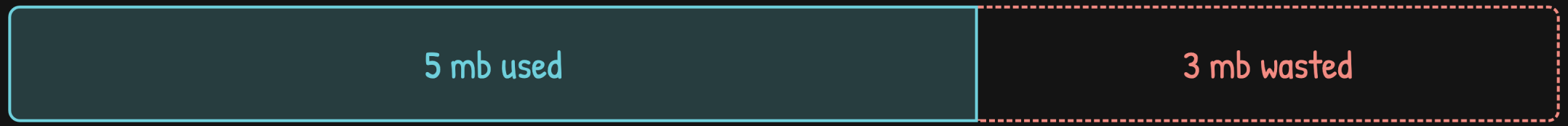
a tiny allocator



power-of-two classes · per-class free lists · o(1), no coalescing, no searching

the allocator's cost

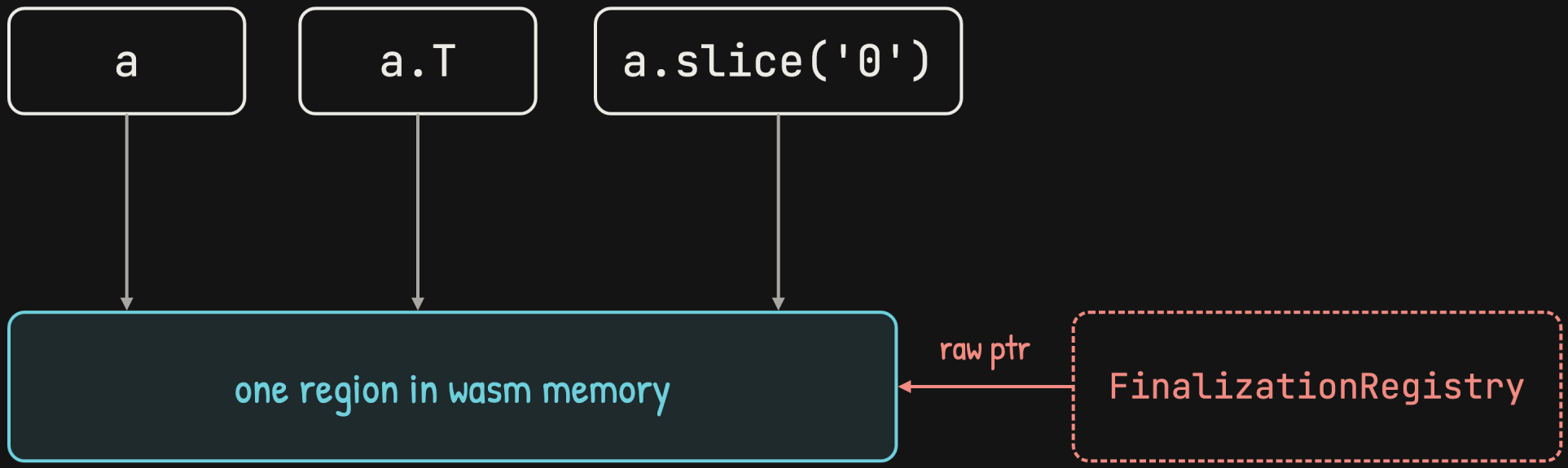
a 5 mb array takes an 8 mb block



no coalescing: freed blocks can't serve another class

so the js fallback arrives sooner than you'd expect

lifetimes



views share a region · registered once per region, not per view

```
{
    using x = np.zeros([1024, 1024]);
    using y = x.multiply(2);
    // …
} // freed here, not at some future gc
```

deterministic cleanup, when you want it

where numpy still wins

fft	sorting	indexing
random	complex dtypes	**large matmul** 512×512: ~4x faster in numpy

methodology

numpy 2.3.1	numpy-ts
openblas 0.3.29	single thread
pthreads, multi-threaded	node 24

16-core m4 max · each side in its own runtime · medians of three runs

numpy-ts v1.7, today

1.36x

geomean vs numpy, across 10,500 benchmarks

building with wasm? ask:

1. where do my bytes live?
2. how many times does each byte cross the boundary?
3. is my op compute-bound or bandwidth-bound?

native speed wasn't about where the math ran.

it was about where the data lived.

thank you

```
npm i numpy-ts
```

github.com/dupontcyborg/numpy-ts