A Primer on Just-In-Time (JIT) Compilation

With a bias towards applications to scientific computing

Dr. Kolen Cheung, Research Software Engineer
Research Software & Analytics Group, University of Exeter

August 5th, 2026

Abstract Numba, JAX and Julia are the three just-in-time compilers most likely to be found in production scientific computing. Run the same small problems through all three and what separates them turns out not to be speed: each language design comes bundled with a different set of abstraction layers — its own tower of intermediate representations, its own introspection tools, its own notion of what you are allowed to say — and the optimisation strategy is baked into that bundle rather than chosen independently of it. The useful question about any of them is never "is it fast?" but *which language am I actually writing?* — and the answer is never the one in the file extension. Every speed-up below is a rewrite that computes exactly the same thing: a compiler pass performed by hand. What the three differ in is how much of that narrowing they leave to you.

Each system is walked down its own tower, with a runnable notebook behind every figure: why fusing four array allocations into two is where the speed actually comes from; why naming an intermediate costs Numba a 127 MB round trip that JAX's tracer never pays, and why hand-writing the loop then beats array notation; why Julia's fusion is syntax you write rather than an inference the compiler makes, why attaching physical units can cost exactly nothing, how a routine `ustrip` on a symmetric sparse matrix silently densified a matrix that would have been 450 TB, and what it means that a language is its own metaprogramming language. Then what the layering itself buys — separation of concerns, a place to verify, a place to optimise, a place to express intent — what compiling late adds on top of it, and what it costs: no artefact to archive, no code to sign, verifiability traded for performance.

Then a digression — an LLM read as a JIT compiler: the ultimate late binding and the ultimate metaprogramming language, but performing enormous narrowing with no specification licensing it and no guard checking it. Bun's 535,000-line Zig-to-Rust rewrite worked because it had both.

The slides, this write-up and the PDF are generated from a single source, so everything that was on screen is here, together with what was said around it.

Table of contents

Before we start 2
- Which language am I actually writing? 2
- You already run a dozen JITs 3

Introducing Numba 3
- A decorator, and a tower underneath it 3
- Why write the loop? Fusion stops where you named a value 4
- `objmode`: `jit` always compiles; it does not always help 5

Introducing JAX 6
- Tracing, not typing 6
- Write the maths, get the loop 7
- Numba vs. JAX on a real kernel 8

Introducing Julia 8
- No decorator — the whole language works this way 9
- Julia's type system & multiple dispatch 9

Julia is its own metaprogramming language 11
Julia's superpower — and its price 12
Stepping back: the tower of abstractions 13
Numba vs. JAX vs. Julia: a high-level comparison 13
The tower of abstractions 14
Lowering through the tower 14
Lowering through the tower, in the case of JIT 14
What "later" knows that "earlier" can't 15
What "later" costs 15
Trust 15
Reproducibility and provenance 15
Signing and JIT are in irreducible tension 15
Digression: LLM as a JIT compiler 16
The analogy: why an LLM is good 16
Where the analogy breaks 16
The tower loses its direction 16
Example: Bun, transpiling from Zig → Rust 17
Why an LLM is bad 17
Trusting and verifying 17
Wrapping up 18
Takeaways 18
Discussion 18

Slides to this talk: blog.kolen.dev/RSE/UoE/2026-08-05-jit.html

This is a companion write-up to a talk I gave at the RSA technical catch-up on 5 August 2026. A while ago I ran a poll for the journal club and this was the topic that came out on top, so here it is — as a technical catch-up rather than a journal club, since there is no obvious paper to read.

I have talked about two of the three tools here before; see the Numba vs. JAX case study from PyCon UK 2025. Julia is the new one, carried in from a project we recently finished: a just-in-time compiled language dedicated to scientific computing.

The slides and this write-up come off a single source, so everything that was on screen is below. The prose around it is what I said on the day, cleaned up — and where I ran out of time, which was most of the middle section and nearly all of the last one, it is what I would have said, written from the slides. Every code sample has a runnable notebook behind it, linked at the top of its section.

Before we start

Two things before the tools themselves: the question everything below hangs off, and a list of the JIT compilers already running on your machine.

Which language am I actually writing?

Not the one in the file extension. Who should have written it: me, or the compiler?

The answer is never the one in the file extension. Every speed-up below is a rewrite that computes exactly the same thing as the version before it — same language, same file, same result — which makes it a compiler pass, performed by hand, in the source layer, because that was the only layer I was allowed to stand on. So the question is really: who should have written it, me or the compiler?

Put plainly: the different optimisations I show below within a single language are not performance tips. They are the demonstration that **the abstraction is leaking**: you are standing in for the compiler, transforming one function into another that computes the same thing and runs at a different speed, because the compiler sees the two differently.

Seen that way, Julia and JAX are powerful for opposite reasons. Julia's metaprogramming lets you program the compiler, so instead of transforming the code by hand you build the abstraction that lets the compiler do the transforming. JAX goes the other way: it is restrictive enough that hand-transforming is not on the table, so you write it idiomatically and the compiler does the rest. Neither is free — Julia still needs you to find the transformation first,

and JAX's restriction only pays inside the domain it covers — but Numba does neither, which is why it is where the leak shows most plainly.

Numba, JAX and Julia are three different answers to that — not three speeds. Each comes with its own tower of abstractions, its own restricted dialect, and its own idea of what you are allowed to say, and the optimisation strategy arrives bundled with the rest rather than being something you pick separately.

So: one long section of concrete examples, three JIT compilers taken one at a time; then a step back into the more abstract way of talking about them; then, at the end, the surprising bit.

You already run a dozen JITs

- **Browsers.** Practically every JS engine in use is a JIT.
- **Databases.** PostgreSQL (`jit=on`, LLVM-based, since PG11); Spark whole-stage codegen.
- **CUDA itself.** PTX → SASS at load time — which is why the first kernel launch is slow, and why `~/.nv/ComputeCache` exists.
- `torch.compile` — Dynamo rewrites bytecode → Inductor → Triton. Ubiquitous in ML; it arrives with PyTorch whether or not you went looking for a JIT.
- **The Linux kernel.** eBPF is JIT-compiled.
- **Your GPU driver**, every time it compiles a shader.
- **Regex.** PCRE2-JIT. **CPython itself**, since 3.13 (PEP 744, experimental, opt-in).

This is not an exotic topic you can opt out of.

CPython's own is the only one on that list you have to opt into — PEP 744, still experimental and off by default. Everything else is running whether you picked it or not.

Introducing Numba

Numba is a Python library. Once you import it you get a decorator, and that decorator completely hijacks the function it is applied to and turns it into something else: it compiles it.

A decorator, and a tower underneath it

Here is a very simple example: a one-line Python function doing some linear algebra, and the change you have to make to get it just-in-time compiled. It is not much of a change — you add a decorator. The signature I have written into it is unnecessary in this case; a bare `@jit` works here. I like being specific.

- Numba-jit can accelerate even a simple NumPy function that should already be quite fast — 1.87 µs → 791 ns here
- The win is not "compiled beats interpreted": it is **operator fusion**
  - memory allocation: 4 in NumPy — `A @ B`, `α · (A @ B)`, `β · C`, and the sum
  - 2 in Numba — `A @ B` and the fused result
- tower of abstractions: CPython bytecode → Numba IR → typed Numba IR → LLVM IR → assembly
- Numba compiles from **bytecode**, not source — the decorator is the seam between Python-as-host-language and Python-as-mini-language

C.f. `intro_numba.ipynb`

```
def mul_numpy(C, A, B, α=True, β=False):
    return α * (A @ B) + β * C

@jit("f8[:, ::1](f8[:, ::1], f8[:, ::1], f8[:, ::1], f8, f8)", nopython=True, nogil=True)
def mul_numba(C, A, B, α=True, β=False):
    return α * (A @ B) + β * C
```

```
mul_numpy   1.87 µs
mul_numba    791 ns      2.4×

new arrays created:  numpy 4,  numba 2
```

The compiled version is about 2.4× faster than the pure NumPy one, and the reason is where the abstraction falls. In Python every operation in that expression goes through an operator, and every operator creates a new object. `A @ B` multiplies the two and hands you back a new array. `α *` that gives you another new array. `β * C` gives you a third. The addition itself gives you a fourth. You need one thing at the end, and you have gone through a lot of memory allocation to get there. Put the decorator on and Numba is able to see through most of it and cut the allocations down.

The one it cannot cut is the matrix multiplication. Numba still does two allocations here, and in principle you only need one. `A @ B` gets dispatched to a linear algebra library, which allocates its own output, and Numba cannot see past that operator to avoid it. (The notebook checks the 4 against 2 in three independent ways: `arrayexpr` nodes in the typed IR, `tracemalloc` peak bytes, and an `ndarray` subclass counting `__array_finalize__`.)

The other thing worth highlighting is that what happens behind the scenes is very complicated, even though all we did was add one decorator. What Numba sees is **not your source code**. CPython has already compiled the function to bytecode, so Numba sees the bytecode first, and it has to be able to understand that bytecode — in some cases manipulating it — to work out what you wrote. Then it keeps lowering: to Numba's own IR, then to a typed Numba IR, then to LLVM IR, then to assembly. It exposes every rung on the way down, through `.inspect_types()`, `.inspect_llvm()` and `.inspect_asm()`, which is what the notebook walks through.

Why write the loop? Fusion stops where you named a value

This one is more complicated, and it is built to show two things: that in Numba you can write loops and still be quite fast, and that here writing the loop is *faster* — which is counter-intuitive to what we are trained to believe when we write array code. In NumPy you optimise by vectorising, by thinking about broadcasting and not using loops. The kernel is a Mexican hat over a grid.

$$\psi_\sigma(x,y) = (1-r^2)\,e^{-r^2/2}, \qquad r^2 = \frac{x^2+y^2}{\sigma^2}$$

- `f_numpy` is apparently the fastest possible NumPy
- `f_numba` — same source, compiled — buys only **1.16×**: 16M points already amortised the interpreter, and this kernel is memory-bandwidth-bound
- Numba's fuser works one `arrayexpr` node at a time, over the array notation *as written* — naming `r2` forces it into memory: **a 127 MB round trip**
- `f_numba_loops` is optimal: hoisted, one allocation, `r2` in a register — you gave up notation and bought the top of the memory hierarchy

C.f. `numba_loop.ipynb`

```
def f_numpy(x, y, σ):
    r2 = (x.reshape(-1, 1) ** 2 + y.reshape(1, -1) ** 2) / (σ * σ)
    return (1.0 - r2) * np.exp(-0.5 * r2)

@jit("f8[:, ::1](f8[::1], f8[::1], f8)", nopython=True, nogil=True)
def f_numba_loops(x, y, σ):
    res = np.empty((x.shape[0], y.shape[0]))
    inv = 1.0 / (σ * σ)
    for i in range(x.shape[0]):
        xi = x[i] * x[i] * inv  # invariant in j — hoisted out of it
        for j in range(y.shape[0]):
            r2 = xi + y[j] * y[j] * inv  # a scalar, in a register
            res[i, j] = (1.0 - r2) * np.exp(-0.5 * r2)
    return res
```

```
                                          ms  vs np  peak allocs
f_numpy             broadcasting, interp.  50.26  1.00×   4.0      —
f_numba             same source, compiled  43.51  1.16×   2.0      2
f_numba_inlined     rewritten for fuser    46.67  1.08×   1.0      3
f_numba_loops       loops, hoisted         39.67  1.27×   1.0      1
```

Adding `@jit` to the same source is only slightly faster, and none of these numbers move very far, because the ceiling here is memory bandwidth rather than arithmetic. The column to read is **peak**, not the time: 4 → 2 → 1 is the structural result, and the two ends of the table follow it. The middle row does not — `f_numba_inlined` halves the peak and is still slower than the plain `@jit` — because peak and allocation count answer different questions. It trades the 127 MB `r2` for two kilobyte-sized temporaries, so it holds half the memory while allocating three arrays instead of two. Only `f_numba_loops` wins on both counts.

`f_numba_loops` is the most optimised version, and what it is doing is a lot of low-level optimisation. First, the loops are explicit. Rather than implicitly assuming everything is an array operation and letting broadcasting sort it out — both inputs are 1D, the result is 2D, so they have to be broadcast along different axes — you write down the `for i` and the `for j` yourself. Then there is hoisting: you spot a computation that would otherwise be done over and over again and you lift it out of the loop. The part involving `x` has nothing to do with the inner loop, so it goes above it, and that is one less computation per element. This is a very typical example of how you would optimise a function in C. You could write exactly the same algorithm in C and it would look very similar.

Now, why does the naive `@jit` version still allocate twice? Both of those lines — `r2 = ...` and the return — are individually understood by Numba as array expressions. What it cannot see past is fusing the two together. `r2` gets its own memory allocated, and then that `r2` is used in the second line for the next computation. Ideally you want the intermediate gone, and in the loop version it is; but the compiler, looking at the first function, does not see past the array expression. So `r2` becomes a 3600×4400 array, and that is a 127 MB round trip through memory that buys you nothing.

Two questions came from the room here.

"Does that mean it can see into the exponential function?" It stays a call — it is not going to be expanded and lowered and optimised further. But that is not what this is about. The point is whether it can see that all of this is an elementwise operation. Expand that second line and it is `for i, for j`, repeating the same expression and putting out an entry, every element independent of every other. That is what a compiler ought to be able to see and act on, and if it does, it knows it never needed to create the intermediate `r2` array at all. Written the way it is written, the compiler cannot understand that. Could you write a compiler that sees past it? Yes, actually — there is an example a couple of sections down. Just not inside the abstraction Numba has built.

"Is reducing memory traffic mainly where Numba's optimisation lies, or are there other places?" The way I would put it is that **Numba is almost C, with some OpenMP-like directives on top.** So you optimise a Numba function by thinking like a C programmer, which is what is happening here: you have to think about your memory access pattern, and interchanging the two loop orders changes the performance.

Not because Numba is imitating C, though. Both of them lower to native code for the CPU you are sitting on — Numba through LLVM IR — and at that level it is the memory hierarchy that decides, so any language that gets close to the metal wants the same treatment. What Numba adds on top is that you are not obliged to write everything that way: the array notation is still there, and you drop into loops only where it pays — which also means that, where it does pay, the work is yours.

The leak I described at the start, in its plainest form: `f_numpy` → `f_numba_loops`.

`objmode`: `jit` always compiles; it does not always help

The short version first. Numba has something called object mode, and what it means is that if you put `@jit` on any Python function it will work — but it may work in the way where the compiler says *I cannot optimise this, I am giving it back to the Python interpreter*.

That is the highlight, and it is the general shape of all three of these tools. You build a certain abstraction, which is a small subset of the Python language, and you say: this I understand, and it can be compiled to something very fast; that I do not understand, and it is not part of my abstraction, so I hand it back. The mechanism is a guard: fall off the guard and the more general thing — Python, in this case — handles it.

The longer version, which is what the slide shows. `objmode` is that boundary drawn by hand rather than inferred: a hole through the type system for one named expression, so the rest of the function can stay in `nopython` mode. The example is Kepler's equation solved element by element with `scipy.optimize.brentq`, which Numba has no way to compile.

- `jit` will always work, but it will not always be faster

- When you stay inside a subset of Python (i.e. Numba's dialect), you compile to machine code (language spec → speculation/specialisation)
- When you don't (fall off the guard) → interpreted: always right, but slow
- `objmode` is a **hole punched through the type system**, drawn by hand — it lets the *rest* of the function keep `nopython=True`
- It buys correctness at the boundary, not speed: here the `brentq` call *was* the runtime, so compiling the loop around it changes nothing

C.f. `numba_objmode.ipynb`

```
def kepler_equation(E, M, e):
    return E - e * np.sin(E) - M

@jit
def kepler(M, e):
    E = np.empty_like(M)
    for i in range(M.shape[0]):
        Mi = M[i]
        E[i] = optimize.brentq(kepler_equation, Mi - 1.1, Mi + 1.1, args=(Mi, e))
    return E

def kepler_objmode(M, e):
    E = np.empty_like(M)
    for i in range(M.shape[0]):
        Mi = M[i]
        with objmode(Ei="float64"):
            Ei = optimize.brentq(kepler_equation, Mi - 1.1, Mi + 1.1, args=(Mi, e))
        E[i] = Ei
    return E
```

```
python:   12.63 ms
forceobj: 12.79 ms
objmode:  13.26 ms
```

The three timings are the point, and the point is that nothing happened. What `objmode` buys is a boundary drawn where you can see it: `forceobj` gives up on the whole function at once, whereas this gives up on one named expression and says so in the source.

Introducing JAX

That is a fast introduction to Numba, but there are three of these to get through. Same function as before, the same bit of linear algebra. The decorator looks slightly different, and what happens beneath it is very different.

Tracing, not typing

Numba is fundamentally type-based. Either you spell the types out by hand, as I did above, or you skip it and at call time it works out what the types are and compiles a version of the function specialised to them. JAX is not like that: it is a **tracing** compiler, which in a certain sense is similar and in mechanism is quite different.

You do not specify a type. When you pass an array into the function, JAX substitutes a stand-in for it: a `Tracer`, which carries no data at all, only an abstract value — a `ShapedArray`, which is to say a shape and a dtype and nothing else. Then, because the `Tracer` has gone in, the function actually gets called and runs, just as it would on any other object. Drop a `print` in the body and you can watch it happen. The tracing compiler is observing exactly that: the `Tracer` goes through all the logic in the function, every array operation it meets gets recorded, and the recording is the program.

So it is even more abstract than Numba. It does not see your source code, it does not even see your bytecode, it does not see how you wrote the logic — it traces the *effect* of the function. From there the representation goes to a jaxpr, then to StableHLO, then, depending on the architecture, in this case CPU, to LLVM IR and assembly. The windows

onto it are `jax.make_jaxpr` and `.lower().compile().as_text()`, at the same altitude as Numba's `inspect_types` and `inspect_llvm`.

- Numba type-specialises **bytecode**. JAX **traces**: it runs the function once on abstract stand-ins (`Tracer`s) and records every array op that fires — Python only *generates* the recording
- tower of abstractions: tracing → jaxpr → StableHLO → fused HLO → LLVM IR (CPU) → assembly
- Caches on **abstract shape and dtype** (a `ShapedArray`), not values — new shape, new trace, new compile
- **JAX is not automatically faster.** At `16×12×32` it *loses* to NumPy — that is per-call dispatch (pytree flatten → cache lookup → PjRt launch), not arithmetic. At `1024³` all three land within a factor of two, and which wins is run-dependent

C.f. `intro_jax.ipynb`

```
@jax.jit
def mul(C, A, B, α=True, β=False):
    return α * (A @ B) + β * C
```

```
              16×12×32          1024³
  numpy        1.86 µs         4.56 ms
  numba         744 ns         2.49 ms
  jax          6.17 µs ←worst  2.79 ms
```

XLA still finds the same fusion Numba's `arrayexpr` optimiser found, from a completely different starting representation — `ROOT %multiply_add_fusion`. Fusion is not what is being measured here.

The interesting thing about JAX, which I have mentioned before, is that it specialises not only on type but also on shape. Feed a 1D float64 array of 1024 elements into a function, then one of 4096, and it will compile a new version for the second. That is where a lot of its advantage in accelerating a function comes from.

Not here, though. Compare this against what we had before and the JAX case is actually much slower — the numpy and numba figures are the same benchmark from a few slides ago, re-run, which is why they are close to those numbers without being identical. Change the array size and it comes out differently: at `1024³` I would not read "best" too hard, because it depends on the run and benchmarking is not that repeatable, but all three are in the same ballpark. This function is also simple enough that you cannot optimise it much further than it already is. In the PyCon UK talk I showed plenty of cases where JAX *is* much faster — tens of percent, sometimes considerably more, precisely because it can see the shape too.

That prompted the right question from the room: so where is the overhead? Two places. First the tracing and compiling, which you pay on the first run. Second, because it is now dispatching on type *and* shape, there is a dispatch mechanism to go through: you arrive with some input, it has to find which compiled, specialised version of the function to run, and if that version does not exist yet it has to hand the work back to the compiler. So it is doing heavier dispatching than Numba's thin typed wrapper, and it is built for a different regime — kernels big enough, or repeated often enough, to pay that back.

Write the maths, get the loop

Back to the Mexican hat, unchanged, this time given to JAX.

- Tracing erases the *name* `r2` before fusion ever runs — so XLA fuses straight through it, and picks fusion boundaries by cost rather than by which Python names appeared
  – the exact thing that cost Numba a 127 MB round trip
- Pure functional, fixed shape, static size: **a tighter spec buys more aggressive optimisation**
- No low-level control (c.f. Numba's fastest version) — you cannot hand-write the loop
- For simple algorithms, just write the maths: four lines get what Numba needed ten hand-written ones for
- The cost moved, it did not vanish: retracing per shape, and no data-dependent control flow or mutation inside the compiled region

C.f. `jax_loop.ipynb`

```python
@jax.jit
def f_jax(x, y, σ):
    r2 = (x.reshape(-1, 1) ** 2 + y.reshape(1, -1) ** 2) / (σ * σ)
    return (1.0 - r2) * jnp.exp(-0.5 * r2)
```

Two fusions for the whole kernel — one scalar, one grid:

```
%multiply_divide_fusion      = f64[]           fusion(...)
%exponential_multiply_fusion = f64[3600,4400]  fusion(...)
```

Both squares, both broadcasts, the add, the divide, `1 - r2`, `-0.5 * r2`, `exp`, the final multiply — every one of them, in a single pass. `r2` never exists as an array.

This is the compiler I said existed, the one that can see past a named intermediate. Tracing throws the *name* `r2` away before fusion ever runs, so by the time XLA gets the program there is no name left for fusion to stop at, and it picks its boundaries by cost instead. That is the 127 MB round trip, gone — for structural reasons, not because one fuser is better written than the other. A tighter specification buys more aggressive optimisation.

The price is everything Numba's fastest version had. There is no JAX equivalent of `f_numba_loops` and no way to write one, so for a kernel this simple you get the good outcome for free — write the maths, and the compiler does what I had to do by hand — but the cost moved rather than vanished. You pay it in retracing on every new shape, and in what you are no longer allowed to write inside the compiled region: no data-dependent control flow, no mutation. Branch on a traced value and you get a `ConcretizationTypeError`, which is what `jax_fail.ipynb` is about.

Numba vs. JAX on a real kernel

Everything above is a toy, built to make one point each. The one production kernel I have proper data for is the interferometer curvature calculation from PyAutoLens, which I benchmarked both ways for the PyCon UK talk.

- Same function, two implementations, from the Numba vs. JAX case study: `w_tilde_curvature_interferometer_from` in Numba vs. JAX
- What that study actually concluded:
  - "Porting Numba to Numba" was faster in many cases — **the first draft was not the opponent**
  - The only fair fight is single-CPU-core
  - **Which algorithm wins flips with input size**, even across Numba and JAX
  - ⇒ keep all implementations → profile → pick the best, per science case per system
- The general form, and the one thing to take from this slide: **the language comparison you think you are running is usually a comparison of how hard you tried.** Benchmark your own best effort, not your first draft — especially when you are about to conclude something about the language.

I have made the same mistake in the other direction, which is why I keep saying it. In 2021 I deleted my own Numba kernel from TOAST and replaced it with C++ behind pybind11, on the strength of a ~30% speed-up (`toast@a38d1d6`). Going back to it properly for this talk, the Numba version I benchmarked against was carrying 893 MB of temporaries per call that it did not need — `out += weight * array` allocates one, because the subexpression has to be evaluated before the addition can happen — and writing the loop instead removes them in nine lines of Python. Of the rest, a good part is alignment: NumPy hands back buffers that are reliably 16-byte and reliably *not* 64-byte aligned, so every AVX-512 load straddles two cache lines, and on this machine that alone is worth about a fifth of the runtime. The `parallel for simd` I was pleased with peaks at two to four threads and is slower than serial by 32, in either language, because the kernel has an arithmetic intensity of 0.25 and nothing left to overlap.

The swap was still the right call — but for reasons that have nothing to do with the number I put in the commit message. The whole thing is worked through in `numba_vs_pybind11.ipynb`.

Introducing Julia

Julia is very different, because Julia has no `@jit` — Julia itself is the JIT. Every function is compiled to machine code, specialised on its argument types, the first time it is called with a new combination of them. There is no decorator because there is no boundary to mark.

No decorator — the whole language works this way

And here is that same function again. It looked much the same in NumPy, in Numba and in JAX, and at first glance it looks much the same in Julia too. Three ways this time.

- Julia has no `@jit`: *every* function is compiled to machine code, specialised on its argument types, the first time it is called with a new combination
- In Julia, **broadcast is JAX's fusing equivalent** — lazy, better memory access pattern — but it is *syntax you write*, not an inference the compiler makes
- tower of abstractions: lowered IR → typed IR → LLVM IR → assembly
  – and the fusion decision happens at **lowering**, before any type is known
- See how `A * B` and `mul!` become an opaque BLAS call — the Julia compiler, including all its lowering, cannot see past it

C.f. `intro_julia.ipynb`

```
function mul_naive!(C, A, B, α = true, β = false)
    C .= α * (A * B) + β * C
    return C
end

function mul_fused!(C, A, B, α = true, β = false)
    C .= α .* (A * B) .+ β .* C
    return C
end

function mul_native!(C, A, B, α = true, β = false)
    return mul!(C, A, B, α, β)
end
```

```
impl                time       memory   allocs
mul_naive!     633.376 ns    16.34 KiB       12
mul_fused!     277.267 ns     4.09 KiB        3
mul_native!    103.032 ns      0 bytes        0
```

Tidy it up a little and you get those odd dots, which say you want the operation broadcast — and broadcast is the equivalent of what is called fusing in JAX, which I did not properly define earlier, so let me do it here. Imagine you want to go through an array, `for i in something`, and you want to do it twice: in the first loop you add one to each element, in the second you square it. Fusing means combining those two loops so that you do not have to go through the whole of memory again. That matters when the kernel is memory-bound, which it usually is, because otherwise you keep streaming the array in from memory and going back through it. In JAX it is what made `r2` disappear. Here you get it with the broadcasting operator, which has the same effect: do not walk this array over and over creating new intermediate allocations every time.

So the naive function is the first one; do the broadcasting to remove most of the unnecessary I/O and it is more than twice as fast. And Julia happens to have a native function that does exactly this job, so we can throw it in as a third: call `mul!` and it is faster again, which is no surprise: a linear algebra library is specialised for exactly this, and it never needed a temporary at all. It also shows Numba's BLAS boundary from the other side: `A * B` and `mul!` both become an opaque library call that the Julia compiler, lowering included, cannot see into.

And it is the same story here as everywhere else: you have layers of abstraction in Julia too, with `@code_lowered`, `@code_typed`, `@code_llvm` and `@code_native` as the windows. The one thing worth noticing is that `@code_lowered` already shows the fusion decided, before any type is known. Nothing in the compiler chose it. You did, with the dots.

Julia's type system & multiple dispatch

This next one introduces something fairly unique to Julia. It has a type system, which plenty of languages have, and it has **multiple dispatch**, and combining the two is what makes it powerful. The illustration is Unitful.jl, which, as the name suggests, puts units on your numbers.

- Type system illustrated by Unitful:

  - the unit is encoded in the **type**
  - a Unitful `Quantity` and a `Float64` (both `<: Number`) share identical lowered IR, and near-identical typed IR that compiles away to nothing
  - so units cost once, at compile time: **1.443 ns vs 1.443 ns** per call
- Also many array types: dense, sparse, symmetric, tridiagonal… Joy for a mathematician!
- Julia's type hierarchy makes unrelated libraries compose automagically — but when it stops working, it feels like magic too →

C.f. `julia_unitful.ipynb`

```
kinetic_energy(m, v) = 0.5 * m * v^2

kinetic_energy(2.0, 3.0)            # 9.0           1.443 ns
kinetic_energy(2.0u"kg", 3.0u"m/s") # 9.0 kg m^2 s^-2  1.443 ns
```

```
julia> typeof(ustrip.(u"MHz", Su))        # sparse + Unitful
SparseMatrixCSC{Float64, Int64}           # fine

julia> typeof(ustrip.(u"MHz", Symmetric(Su)))
Matrix{Float64}                           # densified!
```

```
dense:  3.614 ms
sparse: 7.946 µs      455× at N = 4000, 450 TB at production size
```

- magic: methods for those types are already defined
- broken magic → **missing methods**; nobody foresaw *this* combination
- it changed the **exponent**, and nothing in the program *said* so
- but the fix is always local: write the one method

Take a function calculating kinetic energy. Call it with plain numbers and you get a number. Attach units and you get a number with units — and note in passing that the `u"kg"` syntax is itself a bit of metaprogramming, a library inventing new syntax, which is the subject of the next section.

Attaching units to numbers is not new; you can do it in Python. What is interesting here is that a Unitful `Quantity` is **a new type**, and for most practical purposes, as far as the compiler is concerned, it is a `Float64` carrying a compile-time tag, so it has no runtime cost — and I should nail down that piece of vocabulary, because it trips people up. In a just-in-time compiled language, **compile time is the first time the function gets run; every subsequent run of that function is runtime.** So at runtime it costs nothing: have a function you call a million times and it is free.

One way to see the effect is to look at the IRs. For a function as simple as this one, the lowered IR of the two calls is *identical*. The next one down, the typed IR, is near-identical — not exactly the same, because you have a different type. Go further down and the LLVM IR is mostly the same too. That is the proof that there is no performance difference, and the measurement agrees: 1.443 ns against 1.443 ns. The only overhead you might think about is the flip side, that the next time you throw in a new unit you have to compile the function again. Depending on the use case that is free, or it is expensive if you keep changing units all the time.

The other thing I want to mention is that Julia's array types are just as rich, and I liked that before I had even started using the language: you can have dense matrices, sparse matrices, symmetric matrices, tridiagonal matrices, and so on. All of those are types, so you can *expect* a matrix to be symmetric, and because of multiple dispatch you can handle each of them differently. That is where composability starts to feel magical: two libraries that have never heard of each other compose, because the methods for those types are already defined. Define a new number type — a quaternion, say — and a differential equation library that only ever knew about real and complex numbers may well just work.

The reason I bring it up is that on the project we ran into a problem here, and it surprised me. I had a sparse matrix, and I did something called `ustrip`, which just removes the units and gives you back the bare numbers —

about as ordinary an operation as there is. Apply it to a sparse matrix and you get a sparse matrix. Apply it to a *symmetric* sparse matrix and what comes back is dense.

Fortunately I caught that before it went anywhere near production, because it would have been disastrous: these sparse matrices get really, really big, and if I calculated correctly there is no single node that could hold the dense version in memory. It is not a slowdown, it is a program that does not run. And the thing to notice is that it changed the exponent and nothing in the program said so — no error, no warning, no annotation anywhere that changed. The same class of thing bit me trying to feed Unitful quantities to a differential equation solver, which also would not work.

The reason, without going too far into how the language works, is that you still have to implement the methods. Think of it like Python classes: subclass something and you have a subtype, and sometimes the methods keep working automatically, but sometimes what you did to the subclass means that method has to change too. It is similar here. You may need to define a new method to handle that particular case, and it is an explosion of combinatorics — there is some unforeseen combination of things you did for which no method exists to help you. The composability is real and so is the explosion, and they are the same mechanism. The fix is always local, one missing method; the hard part is noticing.

Julia is its own metaprogramming language

Metaprogramming is a key selling point of Julia.

We did not put it this way when we introduced Numba and JAX, but once you have seen those examples you can view Python, in both cases, as **the metaprogramming language** — and the Numba or JAX function you are compiling as **the mini language**. That is not typical. Metaprogramming in C is macros: a very small language sitting on top of a big one that does all the actual work. In Numba and JAX it is the reverse. You have a very small language doing a subset of what Python can do, and a big metaprogramming language, Python itself, above it. Anything you cannot do in Numba or JAX, you metaprogram in Python.

In Julia that boundary disappears, and in that sense I think it is the best JIT for scientific computing. You have a full and quite expressive language — bigger than C — which is also its own metaprogramming language. All code is data.

The example evaluates a polynomial, and the shape of the problem is worth thinking about. For a polynomial of degree n you have $n + 1$ coefficients. You can throw in any `x`, and `x` is what keeps changing between calls; the coefficients are usually constants, and the degree may differ between call sites — I might want a degree-100 polynomial next. Write the function the ordinary way and the number of coefficients is unknown until runtime, which has a performance implication. Can you optimise that away? Yes, and Julia gives you two routes to it, at two different points in the pipeline.

- In Numba and JAX, Python is the *host* language and the jitted subset is an *embedded* one. In Julia there is no such boundary: **the language is its own metaprogramming language**
- All code is data, and you can manipulate it
- Two different points in the pipeline reach the same machine code:
  - **type inference** resolving a chain of dispatches (`NTuple` recursion), or
  - **a macro** rewriting the source before names are even parsed into scopes
- Both unroll the loop over coefficients away; both land ~5.3× faster than the runtime version

C.f. `julia_as_metaprogramming_lang.ipynb`

```
# baseline: the inner loop over `coeffs` survives to runtime
for c in coeffs
    acc = acc * x[i] + c
end
```

```
# 1. unrolled by type inference: Tuple{} vs Tuple are different types
horner_step(acc, xi, coeffs::Tuple{}) = acc
horner_step(acc, xi, coeffs::Tuple) =
    horner_step(acc * xi + first(coeffs), xi, Base.tail(coeffs))
```

```
out[i] = horner_step(0.0, x[i], coeffs)  # coeffs::NTuple{N,Float64}
```

```
# 2. unrolled by the macro, before lowering ever runs
macro horner(x, coeffs...)
    acc = :(0.0)
    for c in coeffs
        acc = :($acc * $x + $c)
    end
    return esc(acc)
end

@horner(x, 1.0, -3.0, 2.5)  # ⇒  ((0.0 * x + 1.0) * x + -3.0) * x + 2.5
```

```
implementation                  median time         memory   speed-up
runtime (Vector arg)              2204.6 μs     8000072 B     1.00x
recursive (NTuple dispatch)        414.6 μs     8000072 B     5.32x
macro (@horner)                    419.4 μs     8000072 B     5.26x
```

The macro is the one I talked through. You define it so that when it is called it expands into an expression with the coefficients already in it as known constants. *That* is what goes to the compiler, so the compiler gets to assume the coefficients are constant, and the unrolling of the loop happens for free — look at the expansion and there is no loop left, just one big expression. There is no inner loop left over an unknown number of coefficients.

The other route reaches the same place from the type system rather than from the syntax. A `Tuple{}` and a non-empty `Tuple` are different types, so writing `horner_step` as a pair of methods makes the recursion terminate by dispatch, and type inference unrolls the chain at compile time. Two entirely different mechanisms, the same machine code, the same ~5.3×. Memory is identical across all three versions — one `similar(x)`, nothing incidental — so the whole of the difference is in the arithmetic each one generates.

None of which is exotic: Base Julia ships exactly this as `evalpoly`/`@evalpoly`, one function covering both mechanisms depending on whether the coefficients arrive as a tuple or as macro arguments. For contrast, the same idea in Python takes about thirty lines of descriptor-and-proxy machinery and still leaks through the object model in several places.

Julia's superpower — and its price

The claim, in one sentence: open generic functions + multiple dispatch + parametric types + aggressive specialisation + accessible compiler infrastructure + metaprogramming makes it possible to build **high-performance, composable abstraction layers over heterogeneous hardware**.

- Multiple dispatch provides **composability**; specialisation provides **performance**; metaprogramming provides **syntax**; compiler extensibility provides **new targets**.
- Everywhere else in this talk, a leaky abstraction leaves one move: **narrow by hand, at the source, once per call site.** Julia lets you put the transformation where it belongs — `@horner` is a compiler pass *with a name*; `evalpoly` is that pass in the standard library.
- **The price:** the seam between the language and the compiler is gone. You cannot tell by reading which one you are looking at — the same missing seal that let `ustrip` change the exponent.

- Why We Created Julia
- Reactant.jl — Julia → StableHLO/XLA
- JACC.jl

- GPU backends via `GPUCompiler.jl`: CUDA, AMDGPU, oneAPI, Metal
- KernelAbstractions.jl — write once, run anywhere; Julia's answer to what JAX answers with XLA

This last one has no example on it; it is the design of Julia rather than a demonstration of it. Open generic functions: the same function can behave completely differently depending on what you give it, because when you implement a new method with different argument types, multiple dispatch sends the call there. Change the type of an argument to `mul` and you dispatch to whatever method someone wrote for symmetric matrices. Add aggressive specialisation,

and accessible compiler infrastructure — you can program the compiler, because Julia is its own metaprogramming language — and what you get is the power to build more abstraction over what the language can do. That is not something we saw in the case of Numba or JAX.

The one-sentence claim at the top is my own summary, not a quotation from any of the links under it. *Why We Created Julia* is the original post that launched the language, and it is still a good read; I liked it even before I used Julia. The message is essentially: we are greedy, this is what we want, and that is why we made it — a great many things at once, in one language. Reactant.jl is the one to notice next to it. JAX is powered by XLA behind the scenes, and Reactant compiles Julia functions to target XLA, so if you go that route some of the power of JAX and XLA can benefit your function too — you write the maths and let the compiler do the rest. The GPU backends are all built on `GPUCompiler.jl`, and KernelAbstractions.jl and JACC.jl are the write-once-run-anywhere layer above them. All of it exists because the compiler is programmable.

That resolves something left hanging back in the Numba section. When an abstraction leaks — when you have to write the host language differently to convey an intent the compiler could not otherwise act on — everywhere else here you get exactly one move: narrow by hand, in the source, once per call site. `f_numba_loops` is that move, and it is a compiler pass with no name, one you re-derive at every site and nobody else can reuse. `@horner` is the same pass with a name, written once; and `evalpoly` is what it looks like once the pass has graduated into the standard library.

The price is the previous section seen from the other side. The seam between the language and the compiler is gone, so you cannot tell by reading which of the two you are looking at: `@horner(x, ...)` looks like a function call and is a compiler pass, while `ustrip.(u"MHz", Symmetric(Su))` looks like one operation and is a lowering decision that changed the exponent. You can build the abstraction properly *because* the layers are not sealed, and it fails silently for exactly the same reason.

Stepping back: the tower of abstractions

This is the part that generalises past the three tools: what the layering itself buys, what compiling late adds on top of it, and what that costs.

Numba vs. JAX vs. Julia: a high-level comparison

	Numba	JAX	Julia
Backend	LLVM	XLA	LLVM
Targets	CPU; `numba-cuda` is a *second implementation*	CPU, GPU, TPU — one source	CPU; all 4 GPU vendors, one source, via packages
Paradigm	C-like: loops, mutation	functional; purity *enforced*	a full language
You write	a subset of Python + NumPy	a duck-typed NumPy + SciPy	Julia
Closes the leak by	nothing — you hand-narrow	restricting the language	programming the compiler
Recompiles on	**types**	types **and shapes**	**types**
Secret weapon	smallest diff from working NumPy	`jax.grad` — AD as a compiler pass	dispatch + metaprogramming

What you *can* say, what is *fast*, and what is *idiomatic* are **three different sets**.

If someone asks which to use, the answer is on the "you write" row. The optimisation strategy is baked into the language design; you do not get to choose it independently of the programming model.

One caveat on the targets row, because the table has to be terse. Julia out of the box compiles for the CPU you are on and nothing else; the four GPU vendors come from the packages in the previous section — `CUDA.jl`, `AMDGPU.jl`, `oneAPI.jl`, `Metal.jl`, all built on `GPUCompiler.jl`, with `KernelAbstractions.jl` over the top if you want one kernel to run on all of them. It is still one source, which is the contrast with `numba-cuda`, but it is not free of choices.

The line under the table is the general form of that row, and a language decides how much those three sets overlap. Every section above is an instance. In Numba *everything* compiles — that is what `objmode` and `forceobj` mean — but only a subset of it is fast, and that subset is unidiomatic Python: `f_numba_loops` is C spelled in Python. In JAX

rather less compiles at all, but what compiles is fast and it is also the idiomatic thing to have written; you wrote the maths. In Julia nearly everything compiles and most of it is fast, but the sets come apart *silently* — `ustrip` on a `Symmetric` sparse matrix was idiomatic, compiled fine, and changed the exponent.

That is the answer to the question I opened with. You narrow by hand exactly as far as those three sets fail to overlap.

The tower of abstractions

- Source code → AST → IR (usually *several*) → assembly → machine code.
- Note the plurality: Numba alone goes CPython bytecode → Numba IR → typed Numba IR → LLVM IR → MachineIR → x86 — six representations, none of them redundant.
- Each layer is a self-contained abstraction. Leaky, but self-contained.
- Each layer therefore gives you:
  - **separation of concerns** — the layer below doesn't care how you got here
  - **a place to verify**
  - **a place to optimise**
  - **a place to express intent** — the high level carries project/author intent and correctness, in the way a mathematical proof is structured: layers built on layers
- Hence: different languages, compilers, and libraries are *tradeoffs between these abstractions*, not merely differences in syntax or speed.

Every section above walked the same staircase, and the plurality of intermediate representations is the reason each of those sections had a set of introspection tools to walk down — and the reason they kept looking familiar from one tool to the next. `inspect_types`, `make_jaxpr` and `@code_typed` are the same instrument at the same altitude in three different systems.

Lowering through the tower

- Lowering is **lossy in representation** and **faithful in semantics** — semantic narrowing.
  - source semantics does not specify one behaviour; it specifies a *set* of permitted ones
  - compilation narrows that set. It does not step outside it.
- Lossiness → room for optimisation. If lowering had to be reversible you could not constant-fold, fuse, or vectorise at all.
- Different layers of IR → different rooms for optimisation. Some classes are only *recognisable* further up: in x86 you cannot see that you are in the "this is a matmul" class.
- Each layer has a **specification**, and that is what makes travelling down the tower possible.

`f_numpy` → `f_numba_loops` was semantic narrowing — same language, same result, fewer permitted behaviours — and I did it by hand, because Numba's specification was not tight enough to do it for me. The dots in `mul_fused!` are the same thing. JAX does not need either, because it was handed a narrower language to begin with. So the useful question about any of these tools is never "is it fast?" but **"which language am I actually writing?"** — and each of them is an answer to **who does the narrowing**: Numba says you, JAX says the compiler, Julia says you and then hands you the compiler so you can automate yourself.

Lowering through the tower, in the case of JIT

- With JIT you get to **see the data** (narrowing), so the compiler can do more aggressive partial evaluation / specialisation than an AOT compiler can.
- All the JIT compilers we have seen have **different specifications** — and a narrower spec means more room to optimise (e.g. JAX).
- Some specs are leakier than others. In Numba and Julia you often have to write the *host* language differently to convey the same intent in a way the compiler can act on (`f_numba_loops`; the dots in `mul_fused!`).
- Two responses to a leak: **patch it at the source**, or **add a rung**. One is a rewrite you repeat; the other is a pass you name once — and only Julia makes the second one routine.
- Ideally the separation of concerns between layers would be total. It isn't, and that gap is where all the practical difficulty lives.

Patch it at the source, or add a rung: that is the general form of the Julia observation above. Only the second response accumulates — a rewrite you repeat leaves the tower exactly as tall as it was.

What "later" knows that "earlier" can't

- Concrete **types** — no boxing, no dispatch, no polymorphism to hedge against.
- Concrete **shapes** — loop bounds become constants; tiling and unrolling become decidable.
- Runtime **values** — things that are constant in this run but not at build time.
- Branch **frequencies** — which path is actually hot.
- **The machine it is actually running on.** An AOT binary shipped to a heterogeneous cluster targets the lowest common denominator. A JIT gets `-march=native` for free on every node.

The last one is the one that matters most where I work. Ship an ahead-of-time binary to a heterogeneous cluster and it has to target the lowest common denominator of every node it might land on; a JIT compiles on the node it is running on, so it gets the right architecture for free. Anyone who has fought `module load` and a wall of architecture flags to get one binary onto a cluster knows what that is worth.

What "later" costs

- **Warmup.** You pay compile time inside the user's wall clock, not yours.
- **Memory and shipping weight.** The compiler is now a runtime dependency.
- **Budget.** A JIT cannot afford the expensive passes an AOT compiler can, because the user is waiting.
- **Opacity.** There is no artefact on disk to inspect, archive, or hand to someone else.
- **HPC papercuts, specifically:** 512 ranks JIT-compiling the same kernel at once against a shared filesystem; where the cache lives (`NUMBA_CACHE_DIR` → which storage tier?); compiling on the login node vs. the compute node.

The last bullet is the one I have actually been bitten by. On an HPC system the compile lag is normally easy to justify, though not simply because the jobs are long: it is that what you are compiling is a long-running numerical kernel, so what you gain in runtime usually outweighs what you spend compiling, by a lot. And it is also where the operational details get in the way. Hundreds of ranks starting at once and all compiling the same kernel against a shared filesystem is not a theoretical problem, which is why the Numba call I ship in TOAST sets `cache=False`: every one of them writing a compilation cache is worse than not caching at all.

Trust

JIT is good, but it also has its downsides, and the two here are the ones that are hard to price: what you can no longer archive, and what you can no longer verify.

Reproducibility and provenance

- Which binary actually ran? You cannot archive it, because it never existed as a file.
- FMA contraction and fast-math change results; JIT decisions change which you get.
- Autotuning is nondeterministic by construction (`cudnn.benchmark=True`, cuBLAS algorithm selection).
- Different node → different ISA → different vectorisation → different reduction order → different last bits.

In a compiled language you compile a binary, and that binary is a thing you can archive. It is one of the reasons people like containers — part of what you are doing is archiving all those binaries, and you can inspect them, run static analysis on them, hand them to somebody else. If everything only happens at runtime, just in time, then you do not get those artefacts. And you do not exactly know what it was doing in *this* instance when you ran it, as opposed to what happened on my laptop.

It is worse than it first sounds, because a compiler will happily optimise your code by specialising on your hardware. Sitting on different hardware can itself change what ends up in the compiled code: a different instruction set, different vectorisation, a different reduction order, different last bits. That is the same problem as the one in my reproducibility talk from last November, seen from the compiler side rather than the environment side.

Signing and JIT are in irreducible tension

- JIT'd code **cannot be signed**, because it does not exist until runtime.
- iOS enforces code signing and W^X, so JIT requires a special dispensation: the `dynamic-codesigning` entitlement, historically granted only to WebKit's JS engine.
- WebKit's "bulletproof JIT" is the mitigation: the JIT region gets a second, writable mapping at a randomised secret address, so the executable mapping is never writable. Apple Silicon adds per-thread W^X toggling.
- Generalise: **runtime code generation trades verifiability for performance.**

The previous section is that problem in our own domain, science and reproducibility. This is the same problem from another angle, which is security. If you cannot look at the generated code beforehand, run your analysis over it and sanitise it, then you have a new question: how do you trust machine code that was compiled a moment ago?

That is a real problem on something like iOS or iPadOS, where you have **W^X** — W for write, X for execute — meaning that a particular region of memory can be either writable or executable, but not both at the same time. Just-in-time compilation does exactly the forbidden thing: read some source or data, generate machine code from it, write that into memory and then execute the memory you just wrote. So on those devices you can almost never run one, with a single exception — the browser's JavaScript engine, which is itself a JIT compiler and gets the `dynamic-codesigning` entitlement to say so. WebKit pays for the privilege with a good deal of machinery, the "bulletproof JIT" among it. The point is that the opacity has security consequences, because you cannot see the binary or sanitise it ahead of time.

It is also why there is no Numba on my iPhone, and would not be even if somebody ported it: generating machine code and then executing it is exactly what the platform will not let a third-party app do.

Everything that follows is a variation on the last bullet.

Digression: LLM as a JIT compiler

The question is: what if you start thinking of an LLM — an LLM *agent*, especially — as a JIT compiler? It runs things just in time, before the program exists. It does not write the eventual code you asked for in one go; it gets there iteratively, at runtime, in your working directory.

The analogy: why an LLM is good

- Now you can see why an LLM is good:
  - the ultimate JIT — it compiles your prompt into code;
  - the ultimate metaprogramming language — you specify intent and abstraction, not mechanism.
- And it is genuinely the same shape: late binding, specialisation from a high-level spec, caching, and a latency/quality dial that behaves exactly like compile-time vs. runtime.

The metaprogramming half is the stronger of the two claims. I do not have to think about the different layers of abstraction between what I want and machine code; I can just tell it my intention. That is my metaprogramming language.

Where the analogy breaks

- A compiler is **semantics-preserving**: there is a source program with a specified meaning, and correctness means the output stays inside the set of behaviours that meaning permits.
- An LLM has no such referent. **A prompt is not a specification** — it does not define a set of permitted behaviours, so "preserving" it is not even a well-formed claim.
- In the vocabulary from the last section, precisely: an LLM performs **enormous narrowing** — from a prompt to one program — with **nothing licensing the narrowing** (no spec) and **nothing checking it** (no guard).
- Compare a JIT: also unsound narrowing, but bounded by a spec above and caught by a guard below.
- **An LLM is not a compiler. It is the first half of one.**

This is the part that most needs saying, because the analogy does not survive contact with it. Everything I like about the comparison is on the previous slide, and all of it rests on a referent that is not there.

The tower loses its direction

- A compiler only goes **down**, because lowering destroys what it would need to climb.
- An LLM has **no preferred direction**, because it is not translating — it is *reconstructing from a learned prior*. It moves:
  - **down** — prompt → code. The classic framing, and the hardest case.
  - **sideways** — Zig → Rust, Python → C++, COBOL → Java, JS → TS.
  - **up** — code → documentation, code → tests, code → a specification. And decompilation, which people are now genuinely doing.
- **It can climb precisely *because* it hallucinates.** Going up requires inventing information that was destroyed. A decompiler cannot invent a good variable name; a model can guess one — and often guesses right, because the prior encodes what humans usually call things.

- One mechanism, two verdicts: what makes it dangerous going down is what makes it useful going up.

Of everything in the digression this is the part I find most useful, because it answers the "but it just hallucinates" objection without having to deny it. Climbing back up the tower means inventing information that lowering destroyed, and invention is the one thing a translator cannot do.

Example: Bun, transpiling from Zig → Rust

Now the example, and the reason I find it interesting. Bun is a JavaScript runtime — the machinery around the JavaScript engine rather than the engine itself, which is JavaScriptCore — and that machinery was written in Zig. In May 2026 it was rewritten in Rust, using a lot of agents. The diff is humongous — far beyond what a human reviewer can do, and nowhere near fitting in the context window of a single model.

- 535,496 lines of Zig across 1,448 files → Rust in **11 days** (3–14 May 2026). https://bun.com/blog/bun-in-rust
- **64 agents in parallel** (4 worktrees × 16), ~50 workflows; 6,502 commits; peak 695 commits/hour. 90%+ automated, one engineer supervising.
- **~$165,000** of API spend, against an estimated 3 engineers × 1 year with no features shipping.
- 128 bugs fixed, ~2–5% faster, several MB smaller binary.

Why it worked — both missing pieces were already there

- the **source program is the specification**: defined semantics, so "correct" means something precise
- the **test suite is the guard** — and it is **written in TypeScript**, so it does not depend on the implementation language: 60,624 tests, **1,386,826 assertions**, zero skipped

…and the tests still were not the specification

- **19 known semantic regressions** got through all of them — each one two languages disagreeing about the spec
- **A test suite is a sample, not a specification**

And it got done. In this case the LLM becomes a transpiler, and what made that possible is the test suite. Bun runs JavaScript, so its tests are written in TypeScript — which means they are independent of the implementation language. Change the language underneath, from Zig to Rust, and it does not matter; you have exactly the same test suite. That becomes the verifiable bit. It is almost a spec, in the compiler sense: something the model can transpile against and check itself off. Not for every valid construct of the language, but for what it actually built.

The second half of the slide is the caveat I would put on my own enthusiasm. Nineteen known semantic regressions still got through those 1,386,826 assertions. For half a million ported lines in eleven days that is a good result rather than an indictment, and the interesting thing is where they landed: in the places the tests were silent about. A test suite is a sample of the specification, not the specification.

Why an LLM is bad

- **The input space is infinite.** No input is wrong. No linting, no static analysis, no abstraction.
- **The output space is infinite.** All outputs are possible — and under agentic use, so is everything that happens *along the way* to producing one.
  - You cannot trust the output (cf. bulletproof JIT).
  - You cannot secure the machine (cf. the lethal trifecta: private data + untrusted content + exfiltration channel).
- **A compiler's blast radius is a process. An agent's blast radius is your network.**
  - Concrete: OpenAI ran a cybersecurity test against an unreleased model with guardrails off. Rather than solve the test, the model broke out of OpenAI's sandbox, then found exploits to break into Hugging Face — in order to steal the answers and cheat on the test. https://simonwillison.net/2026/Jul/22/openai-cyberattack/
- The apparent remedy is a human in the loop. But: would you put a human between the compiler and its output, and ask them to check the machine code?

This is the security section again, taken to its limit. The input space is infinite, so no input is wrong and there is nothing to reject. The output space is infinite, so every output is possible. And under agentic use, whatever the agent decides to do in between is also an infinite unknown space. So you never get to sanitise anything, at any point along the way.

Trusting and verifying

- **Once you can make a problem verifiable — at training time or at run time — an agent starts behaving like a compiler.** Verifiability converts *synthesis* back into *translation*.

- Note the inversion, and it is the whole problem: with a compiler you read the **top** and trust the bottom. With an agent we currently read the **bottom** — the largest, least reviewable representation in the tower — and trust nothing.
- So: **reduce entropy. Shrink the space of things that could come out.** TDD, unit tests, property-based testing, types, contracts, enforced abstractions (Rust without `unsafe`, pure functions, narrow typed interfaces) — all of which were **always** about making behaviour checkable without reading the code. That simply was never their main job before.
- StrongDM's Software Factory: **specs + scenarios drive agents that write code, run harnesses, and converge without human review**. https://simonwillison.net/2026/Feb/7/software-factory/
  - specs and scenarios supply the missing **specification**; the harness supplies the missing **guard** — and it inherits the limit: **you have validated only what the harness can see**

That is the same move as the Julia section, one altitude up. There, the fix for a leaky abstraction was to stop patching at the source and add a rung to the tower. Here the missing rungs are the specification and the guard, and the work is to put them back. Which is why I do not think this is a new discipline: nothing on that list of tests, types, contracts and enforced abstractions was invented for agents, and most of it is already sitting in the repositories we maintain.

Wrapping up

Takeaways

- A JIT is a bet that you will **know more later** — types, shapes, values, profiles, and the actual machine. Sometimes the bet does not pay: JAX lost to NumPy at 16×12×32.
- AOT vs. JIT is a **dial, not a binary**. The question is always *when do you decide?*
- Lowering is **lossy in representation and faithful in semantics**. The lossiness is where all the optimisation lives. The useful question about a JIT is never "is it fast?" but **"which language am I actually writing?"** — and every one of them is an answer to **who does the narrowing**: Numba says you, JAX says the compiler (because you handed it a narrower language), Julia says you *and* hands you the compiler to automate yourself.
- Every performance result here but one came from **removing memory traffic**, not adding FLOPs. And the language comparison you think you are running is usually a comparison of how hard you tried.
- Runtime code generation **trades verifiability for performance.** Always.
- An LLM is the extreme answer to that same question — it narrows everything, from a prompt: maximal narrowing, no specification licensing it, no guard checking it. The work is putting a layer back — and it is work we already know how to do.
- You never delete human verification. You **relocate** it somewhere smaller: assembly → source → spec → a harness you can audit.

If only one line of this survives, let it be the third: the question is not "is it fast?" but which language you end up actually writing, and how much of the narrowing the tool leaves to you.

The one exception to the memory-traffic line is Julia's unrolling. Memory was identical across all three versions there — one `similar(x)` and nothing else — so the whole of that 5.3× is in the arithmetic each version generates.

Discussion

All the code is runnable: Numba · Numba loops · `objmode` · Numba failure modes · Numba parallel · Numba vs. pybind11 · JAX · JAX loops · JAX failure modes · Python as a metaprogramming language · Julia · Unitful · Julia traits · Julia as a metaprogramming language