Performance

Type stability, `@code_warntype`, allocation, benchmarking

Julia's performance model is unusual: the same source can be C-fast or Python-slow depending on whether the compiler can infer types. Nothing warns you when it can't. This section is the checklist for finding out.

The rules, in order of impact

1. **Put code in functions.** Top-level global scope is not optimised at all. A loop in a script and the same loop in a function can differ by 50x.
2. **Type stability.** The return type must be inferable from the argument types. This is the whole game.
3. **Concrete struct fields.** `::Float64` or `::T`, never bare or abstract.
4. **Avoid non-constant globals.** If a function reads a global, its type can change at any time, so the compiler must assume `Any`. Mark it `const`, or pass it as an argument.
5. **Preallocate and mutate** in hot loops instead of allocating per iteration.
6. **`@views` on slices**, `@inbounds` when you're certain.

Everything else is noise until these six are right.

Type stability

A function is type-stable if the type of every variable, and the return type, is determined by the argument types alone.

```
function unstable(n)
    s = 0                # Int
    for i in 1:n
        s += i / 2       # Float64 — s is now Union{Int, Float64}
    end
    s                    # return type depends on whether the loop ran
end

function stable(n)
    s = 0.0
    for i in 1:n
        s += i / 2
    end
    s
end
```

Other common instability sources:

```
# returning different types from branches
f(x) = x > 0 ? 1 : "negative"          # Union{Int, String}

# accumulating into a container of the wrong type
v = []                                  # Vector{Any}
v = Int[]                               # Vector{Int}

# a global
const CONFIG = load_config()            # const, so its type is known
scale(x) = x * CONFIG.factor            # fine
# without const: every call re-checks the type of CONFIG

# hardcoded literals in generic code
function mysum(v)
    s = 0                               # breaks for Vector{Float32}, Complex, Dual, ...
    s = zero(eltype(v))                 # correct
    for x in v; s += x; end
    s
end
```

Small unions (`Union{Int, Nothing}`) are optimised and fine. Large or open unions, and `Any`, are not.

Finding it: @code_warntype

```
@code_warntype mysum([1.0, 2.0])
```

Read the output for red text and `Any` / large `Union` types. The pattern to look for:

```
Body::Union{Float64, Int64}     <- red: unstable return
  s::Union{Float64, Int64}      <- red: unstable local
```

`Cthulhu.jl` (`@descend mysum(v)`) lets you walk down into callees interactively, which is what you need once the problem is two layers deep. `JET.jl` (`@report_opt f(x)`) reports optimisation failures across a whole call tree without manual descent.

```
using JET
@report_opt mysum(v)        # runtime dispatch, allocation-forcing instability
@report_call mysum(v)       # possible errors, method errors
```

Measuring

```
using BenchmarkTools

@btime mysum($v)                    # minimum time over many samples
@benchmark mysum($v)                # full distribution
@allocated mysum(v)                 # bytes allocated — 0 is the goal in hot paths
@time mysum(v)                      # includes compilation on first call; run twice
@elapsed
```

The `$` matters. `@btime f(v)` measures a global lookup of `v` plus the call; `@btime f($v)` interpolates the value and measures the call. Without it your numbers are wrong, usually by a lot.

Profiling:

```
using Profile
@profile expensive_thing()
Profile.print()                     # text tree
Profile.print(format = :flat, sortedby = :count)

using PProf; pprof()                # flamegraph in a browser
using ProfileView; @profview f(x)   # GTK flamegraph
@profview_allocs f(x)               # allocation profiler (1.8+)
```

The allocation profiler is often more actionable than the time profiler, because in Julia most performance problems present as allocations.

Allocation

Every allocation is GC pressure. Common sources and their fixes:

```
# slicing
s = sum(A[:, 1])            # allocates a column
s = sum(@view A[:, 1])      # doesn't

# intermediate arrays in expressions
y = 2 .* x .+ 1             # one allocation (the result) — fusion handles the rest
y = 2 * x + 1               # TWO allocations — no dots, no fusion
@. y = 2x + 1               # zero allocations, writes into existing y

# growing in a loop
out = Float64[]
for x in v; push!(out, f(x)); end        # repeated reallocation
out = similar(v); for i in eachindex(v); out[i] = f(v[i]); end   # preallocated
sizehint!(out, length(v))                 # if you must push!

# closures capturing reassigned variables (boxing)
```

```
function make()
    n = 0
    () -> (n += 1)          # n is boxed
end

# splatting a runtime-length collection
f(v...)                     # slow if length(v) is not statically known

# string concatenation in a loop
s *= x                      # quadratic; use an IOBuffer
```

Small immutable structs and tuples usually do *not* allocate — they live in registers or inline in their parent. `StaticArrays.SVector{3}` is the tool for small fixed-size vectors in hot loops; it's often 10x faster than `Vector` for 3-D geometry.

Bounds checking and SIMD

```
@inbounds for i in eachindex(v)
    v[i] *= 2
end

@simd for i in eachindex(v)         # allow reordering of float reductions
    s += v[i]
end

@fastmath                            # relax IEEE semantics. Know what you're giving up.
```

`@inbounds` on an incorrect index is memory corruption, not an error. Use `eachindex`/`axes` so the bounds are provably correct, and test with `--check-bounds=yes` in CI.

Julia auto-vectorises simple loops; `LoopVectorization.jl`'s `@turbo` goes further, at the cost of a heavy dependency and stricter requirements on the loop body.

Memory layout

- Column-major: innermost loop over the first index.
- Array of structs vs struct of arrays: `StructArrays.jl` gives you the AoS interface over SoA storage, which is usually what you want for numerical work over records.
- `Vector{MyImmutableStruct}` stores the structs inline. `Vector{MyMutableStruct}` stores pointers. That difference alone is often 5x.

Compilation latency

First-call compilation ("time to first X") is a separate problem from run-time speed.

- Keep one long-lived session and use Revise. Restarting is a rebuild.
- `PrecompileTools.jl`'s `@compile_workload` bakes representative calls into the package's precompile cache, so your users don't pay the cost.
- `PackageCompiler.jl` builds a custom sysimage with your dependencies baked in — the right answer for a CLI tool or a slow-to-load analysis stack.
- Excessive specialisation (dispatching on many type combinations, heavy `@generated` use) increases compile time. `@nospecialize` on arguments where you don't need specialised code.

A workflow

1. Write the obvious version. Make it correct, with tests.
2. `@btime` it. If it's fast enough, stop.
3. `@allocated` — is it allocating in a loop it shouldn't be?
4. `@code_warntype` or `@report_opt` on the hot function. Fix instability first; it's usually the whole problem.
5. `@profview` to find where the time actually goes, rather than where you assume.
6. Only then: `@inbounds`, `@views`, StaticArrays, manual unrolling.

Steps 4 and 5 in the wrong order is the standard mistake — micro-optimising a function that's dispatching dynamically achieves nothing.

Quick reference

```
@time, @timev, @elapsed, @allocated
@btime, @benchmark, @belapsed          # BenchmarkTools, with $ interpolation
@code_warntype, @code_typed, @code_llvm, @code_native, @code_lowered
@descend                               # Cthulhu
@report_opt, @report_call              # JET
@profile, @profview, @profview_allocs
GC.gc(); GC.enable(false)              # rarely; measuring GC impact
Base.summarysize(x)                    # deep size in bytes
@inbounds, @simd, @fastmath, @views, @inline, @nospecialize
```