Performance and profiling in Julia

Fredrik Bagge Carlson[1]

[1]Dept. Automatic Control, Lund Institute of Technology
Lund University

Outline

1. Write code with performance in mind (think like a compiler)
2. Profile your code
3. Optimize your code

Step one, put your code in functions

A global variable might have its value and its type change at any point. This makes it difficult for the compiler to optimize code using global variables.

$\Downarrow$

Any code that is performance critical or being benchmarked should be inside a function.

```julia
a = 1
@btime for i = 1:100_000
    global a
    a += 1
end
    1.543 ms
    (100000 allocations: 1.53 MiB)
```

```julia
function foo()
    a = 1
    for i = 1:100_000
        a += 1
    end
    a
end
@btime foo()
    1.270 ns
    (0 allocations: 0 bytes)
```

Avoid global variables (unless declared `const`)

```julia
PRINT = false
function foo()
  for i = 1:1_000_000_000
    if PRINT
      print(i)
    end
  end
end
@time foo()
  0.795711 seconds
```

```julia
const PRINT = false
function foo()
  for i = 1:1_000_000_000
    if PRINT
      print(i)
    end
  end
end
@time foo()
  0.000002 seconds
```

`PRINT` is `false` in both cases, but the compiler can rely on it in the second case

Type declarations, type stability

Useful as assertion for debugging, but does not make the code faster.

Exception: Declare specific types for fields of composite types so that the compiler knows the memory layout

```julia
struct Foo
    field
end
```

```julia
struct Foo
    field::Type
end
```

It is in general bad for performance when the type of a variable can be changed at runtime, type annotation will prevent this.

Type stability

An example of type instability

```julia
stable(i) = rand() > .5 ? 1 : -1

function foo()
    a = 1
    for i = 1:100_000
        a += stable(i)
    end
    a
end

@btime foo()
  162.940 μs
```

```julia
unstable(i) = rand() > .5 ? 1. : -1

function bar()
    a = 1
    for i = 1:100_000
        a += unstable(i)
    end
    a
end

@btime bar()
  684.704 μs
```

In `bar` the compiler does not know ahead of time which + method to call (`Int` or `Float64`) and dispatch happens at runtime.

Type stability

```julia
quasistable(i) = rand() > 1.5 ? 1. : -1

function baz()
    a = 1
    for i = 1:100_000
        a += quasistable(i)::Int
    end
    a
end
```

```julia
@btime foo()
162.940 µs
@btime bar()
684.704 µs
@btime baz()
170.997 µs
```

```julia
julia> @code_warntype quasistable(1)
 Body::Union{Float64, Int64}
```

Now the compiler knows that the type is `Int` after the type assertion, and dispatch can be determined at compile time, even though the return type of `quasistable` is set valued.

Julia uses column major convention

```julia
function foo()
    x = Matrix{Float64}(undef,
                        1000, 1000)
    for i = 1:size(x,1)
        for j = 1:size(x,2)
            x[i,j] = i*j
        end
    end
end
@btime foo()
2.555 ms (2 allocations: 7.63 MiB)
```

```julia
function bar()
    x = Matrix{Float64}(undef,
                        1000, 1000)
    for j = 1:size(x,2)
        for i = 1:size(x,1)
            x[i,j] = i*j
        end
    end
end
@btime bar()
689.623 μs (2 allocations: 7.63 MiB)
```

Think about this when you are choosing how to store your data!

Avoid unnecessary memory allocation

Julia passes arrays as references. Use this to re-use already allocated memory.

```julia
function food()
    A = Matrix{Int64}(undef,100,100)
    for i = eachindex(A)
        A[i] = i
    end
    return A
end

function eat()
    for i = 1:10_000
        chicken = food()
        sum(chicken)
    end
end
@btime eat()
144.466 ms (20000 allocations: 763.70 MiB)
```

New plate every time, lots of time to clean! (garbage collect)

```julia
function beer!(A)
    for i = eachindex(A)
        A[i] = i
    end
end

function drink()
    glass = Matrix{Int64}(undef,100,100)
    for i = 1:10_000
        beer!(glass)
        sum(glass)
    end
end
@btime drink()
54.119 ms (2 allocations: 78.20 KiB)
```

Use the same glass every time, drink beer faster!

Profile your code

Profiling

Profiling

Your goto-tool is always `@btime` from `BenchmarkTools.jl`, watch memory allocation and GC-time

- Type instability
- Allocations
- Do not benchmark in global scope
- Do not benchmark compilation time
- Interpolate global variables `@btime testfun($a)`
- `@btime` runs the expression several times and reports the minimum

Profiling

Julia has built in profiling capabilities

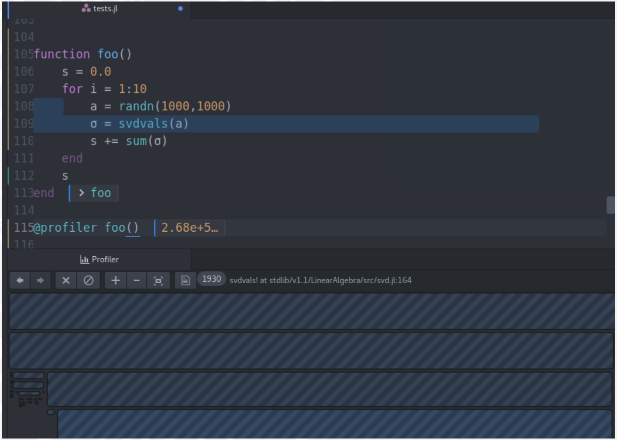
```
julia> @profile foo()
```

```
julia> Profile.print()
      23 client.jl; _start; line: 373
        23 client.jl; run_repl; line: 166
           23 client.jl; eval_user_input; line: 91
              23 profile.jl; anonymous; line: 14
                 8  none; myfunc; line: 2
                  8 dSFMT.jl; dsfmt_gv_fill_array_close_open!; line: 128
                 15 none; myfunc; line: 3
                  2  reduce.jl; max; line: 35
                  2  reduce.jl; max; line: 36
                  11 reduce.jl; max; line: 37
```

Atom profile view

The profile viewer in atom is nicer

`@profiler foo()`



Profiling tools

```
julia --help
--track-allocation=none|user|all Count bytes allocated by each source line
```

Traceur.jl

Traceur is essentially a codified version of the Julia performance tips. You run your code, it tells you about any obvious performance traps.

```
julia> using Traceur
julia> naive_relu(x) = x < 0 ? 0 : x
julia> @trace naive_relu(1.0)
naive_relu(::Float64) at none:1
    returns Union{Float64, Int64}
1.0
```

Benchmarking

- Put your code in functions
- Let the function compile before timing (or use `@btime`)
- Watch out for unexpected memory allocation
- Read the performance tips!

Optimization

Optimize your code

- Write a test before you start optimizing to make sure you are still calculating the same thing
- Use the result of `@profiler`, `@btime`, `@trace`, `track-allocation=user`
- If your code spends 50% doing garbage collection, you can sometimes reduce your running time with *approximately* 50% by better memory management.

Optimize your code

```
function slowfunc(x)
    a = foo(x)
    for i = 1:1000
        b = bar(a, randn(2))
        for j = 1:1000
            c = randn(5)
            b += baz(c)
        end
        a += b
    end
    a
end
```

Where do you start looking?

SIMD

Single Instruction Multiple Data

```
a64 = randn(1000000)
a32 = randn(Float32, 1000000)
function regular_sum(x)
    s = zero(eltype(x))
    for i = eachindex(x)
        s += x[i]
    end
    s
end

@btime regular_sum($a64)
770.580 µs
@btime regular_sum($a32)
750.522 µs
```

```
function simd_sum(x)
    s = zero(eltype(x))
    @inbounds @simd for i = eachindex(x)
        s += x[i]
    end
    s
end

@btime simd_sum($a64)
229.129 µs
@btime simd_sum($a32)
86.757 µs
```

StaticArrays

One of the biggest speed-ups (after choosing the right algorithm) if often to use StaticArrays where available

- Size known at compile time
- Optimized operations
- Stack allocated (as opposed to heap allocated)

Benchmarks for 3×3 Float64 matrices

```
Matrix multiplication                    -> 8.2x speedup
Matrix multiplication (mutating)         -> 3.1x speedup
Matrix addition                          -> 45x speedup
Matrix addition (mutating)               -> 5.1x speedup
Matrix determinant                       -> 170x speedup
Matrix inverse                           -> 125x speedup
Matrix symmetric eigendecomposition -> 82x speedup
Matrix Cholesky decomposition            -> 23.6x speedup
```

StaticArrays

Details

- Size and type hard coded, known at compile time
- `VectorSVector` has same memory layout as `Matrix`

```
using StaticArrays
a  = [randn(3) for _ = 1:1000]
am = randn(3,1000)
as = [@SVector randn(3) for _ = 1:1000]
@btime sum($a)
    27.917 μs (999 allocations: 109.27 KiB)
@btime sum($am, dims=2)
    2.798 μs (7 allocations: 304 bytes)
@btime sum($as)
    759.379 ns (0 allocations: 0 bytes)
```

Misc.

FillArrays.jl Represent special arrays efficiently

`repmat,repeat` If you use these to force your problem into a vectorized form, you need to de-matlabify yourself

`collect(1:10)` You most likely do not need to collect.

Avoid allocating slices

`A[:,i]` allocates and copies data, `@view(A[:,i])` doesn't. (`A[i,:]` might however be worth it.)

Parallel Threading and distributed computing

dot-fusion `R = sin.(exp.(A.^2))` compiles into a single loop

- No temporary arrys
- Single pass over data

```
R = similar(a)
for i in eachindex(a)
    R[i] = sin(exp(a[i]^2))
end
```

Other resources

- I would first and foremost recommend the performance tips section in the manual, it's quite comprehensive and readable: https://docs.julialang.org/en/v1/manual/performance-tips/index.html
- Chris Rackaukas has some tutorials on solving ODEs and PDEs in Julia. He highlights a lot of neat Julia functionality and goes through a lot of performance optimizations that extend also outside the realm of ODEs and PDEs
  https://youtu.be/KPEqYtEd-zY Watch around minute 49 for performance optimization
  https://youtu.be/okGybBmihOE
- An introduction to high performance custom arrays | Matt Bauman
  https://www.youtube.com/watch?v=jS9eouMJf_Y&t=1831s&list=PLP8iPy9hna6Qsq5_-zrg0NTwqDSDYtfQB&index=82

Homework

Monte-Carlo simulation of a bootstrap particle filter

- I provide the baseline code
- My code provides a decent particle filter implementation
- The code is bad from a julia-performance point of view
- Your job is to optimize it
- Optimized code has to be equivalent (do not implement different algorithm)

$$
\begin{aligned}
x^+ &= 0.5x + \frac{25x}{1+x^2} + 8\cos(1.2(t-1)) + w \\
y &= 0.05x^2 + v \\
w, v &\sim \mathcal{N}(0, \sigma_w),\ \mathcal{N}(0, \sigma_v) \quad E(wv^\top) = 0
\end{aligned}
$$

The particle filter

```
for t = 2:T # Main loop
    # Resample
    j = resample(w[t-1,:])
    # Time update
    xp[t,:] = f(xpT,t-1) + σw*randn(1,N)
    # Measurement update
    w[t,:] = wT + g(y[t]-0.05xp[t,:].^2)
    # Normalize weights
    w[t,:] -= log(sum(exp(w[t,:])))
end
```

⋮

The Monte-Carlo simulation

```
particle_count = [5 10 20 50 100 200 500 1000 10_000]
time_steps = [20, 200, 2000]
for (Ti,T) in enumerate(time_steps)
  for (Ni, N) in enumerate(particle_count)
    # Calculate how many Monte-Carlo runs to perform for the current
    # T,N configuration
    montecarlo_runs =
        maximum(particle_count)*maximum(time_steps) / T / N
    for mc_iter = 1:montecarlo_runs
      for t = 1:T-1 # Simulate one realization of the model
        x[t+1] = f(x[t],t) + σw*randn()
        y[t+1] = 0.05x[t+1]^2  + σv*randn()
      end # t
      xh = pf(y, N, g, f, σw0) # Run the particle filter
      RMS += rms(x-xh) # Store the error
    end # MC
```

⋮