Optimizing DiffEq Code

Chris Rackauckas

May 25, 2021

In this notebook we will walk through some of the main tools for optimizing your code in order to efficiently solve DifferentialEquations.jl. User-side optimizations are important because, for sufficiently difficult problems, most of the time will be spent inside of your `f` function, the function you are trying to solve. "Efficient" integrators are those that reduce the required number of `f` calls to hit the error tolerance. The main ideas for optimizing your DiffEq code, or any Julia function, are the following:

- Make it non-allocating
- Use StaticArrays for small arrays
- Use broadcast fusion
- Make it type-stable
- Reduce redundant calculations
- Make use of BLAS calls
- Optimize algorithm choice

We'll discuss these strategies in the context of small and large systems. Let's start with small systems.

0.1 Optimizing Small Systems (<100 DEs)

Let's take the classic Lorenz system from before. Let's start by naively writing the system in its out-of-place form:

```
function lorenz(u,p,t)
 dx = 10.0*(u[2]-u[1])
 dy = u[1]*(28.0-u[3]) - u[2]
 dz = u[1]*u[2] - (8/3)*u[3]
 [dx,dy,dz]
end
```

```
lorenz (generic function with 1 method)
```

Here, `lorenz` returns an object, `[dx,dy,dz]`, which is created within the body of `lorenz`.

This is a common code pattern from high-level languages like MATLAB, SciPy, or R's deSolve. However, the issue with this form is that it allocates a vector, `[dx,dy,dz]`, at each step. Let's benchmark the solution process with this choice of function:

```
using DifferentialEquations, BenchmarkTools
u0 = [1.0;0.0;0.0]
tspan = (0.0,100.0)
prob = ODEProblem(lorenz,u0,tspan)
@benchmark solve(prob,Tsit5())
```

```
BenchmarkTools.Trial:
  memory estimate:  10.81 MiB
  allocs estimate:  100152
  --------------
  minimum time:     3.797 ms (0.00% GC)
  median time:      3.936 ms (0.00% GC)
  mean time:        5.217 ms (15.93% GC)
  maximum time:     12.054 ms (46.25% GC)
  --------------
  samples:          956
  evals/sample:     1
```

The `BenchmarkTools.jl` package's `@benchmark` runs the code multiple times to get an accurate measurement. The minimum time is the time it takes when your OS and other background processes aren't getting in the way. Notice that in this case it takes about 5ms to solve and allocates around 11.11 MiB. However, if we were to use this inside of a real user code we'd see a lot of time spent doing garbage collection (GC) to clean up all of the arrays we made. Even if we turn off saving we have these allocations.

```
@benchmark solve(prob,Tsit5(),save_everystep=false)
```

```
BenchmarkTools.Trial:
  memory estimate:  9.47 MiB
  allocs estimate:  88645
  --------------
  minimum time:     3.273 ms (0.00% GC)
  median time:      3.450 ms (0.00% GC)
  mean time:        4.720 ms (15.14% GC)
  maximum time:     11.733 ms (59.09% GC)
  --------------
  samples:          1057
  evals/sample:     1
```

The problem of course is that arrays are created every time our derivative function is called. This function is called multiple times per step and is thus the main source of memory usage. To fix this, we can use the in-place form to ***make our code non-allocating***:

```
function lorenz!(du,u,p,t)
 du[1] = 10.0*(u[2]-u[1])
 du[2] = u[1]*(28.0-u[3]) - u[2]
 du[3] = u[1]*u[2] - (8/3)*u[3]
end
```

```
lorenz! (generic function with 1 method)
```

Here, instead of creating an array each time, we utilized the cache array `du`. When the inplace form is used, DifferentialEquations.jl takes a different internal route that minimizes the internal allocations as well. When we benchmark this function, we will see quite a difference.

```
u0 = [1.0;0.0;0.0]
tspan = (0.0,100.0)
prob = ODEProblem(lorenz!,u0,tspan)
@benchmark solve(prob,Tsit5())

BenchmarkTools.Trial:
  memory estimate:  1.37 MiB
  allocs estimate:  11752
  --------------
  minimum time:     783.501 μs (0.00% GC)
  median time:      803.491 μs (0.00% GC)
  mean time:        979.536 μs (10.64% GC)
  maximum time:     11.625 ms (89.94% GC)
  --------------
  samples:          5075
  evals/sample:     1

@benchmark solve(prob,Tsit5(),save_everystep=false)

BenchmarkTools.Trial:
  memory estimate:  6.70 KiB
  allocs estimate:  47
  --------------
  minimum time:     348.157 μs (0.00% GC)
  median time:      357.046 μs (0.00% GC)
  mean time:        358.957 μs (0.18% GC)
  maximum time:     6.769 ms (93.85% GC)
  --------------
  samples:          10000
  evals/sample:     1
```

There is a 4x time difference just from that change! Notice there are still some allocations and this is due to the construction of the integration cache. But this doesn't scale with the problem size:

```
tspan = (0.0,500.0) # 5x longer than before
prob = ODEProblem(lorenz!,u0,tspan)
@benchmark solve(prob,Tsit5(),save_everystep=false)

BenchmarkTools.Trial:
  memory estimate:  6.70 KiB
  allocs estimate:  47
  --------------
  minimum time:     1.741 ms (0.00% GC)
  median time:      1.781 ms (0.00% GC)
  mean time:        1.785 ms (0.00% GC)
  maximum time:     4.335 ms (0.00% GC)
  --------------
  samples:          2797
  evals/sample:     1
```

since that's all just setup allocations.

But if the system is small we can optimize even more. Allocations are only expensive if they are "heap allocations". For a more in-depth definition of heap allocations, there are a lot of sources online. But a good working definition is that heap allocations are variable-sized slabs of memory which have to be pointed to, and this pointer indirection costs time. Additionally, the heap has to be managed and the garbage controllers has to actively keep track of what's on the heap.

However, there's an alternative to heap allocations, known as stack allocations. The stack is statically-sized (known at compile time) and thus its accesses are quick. Additionally, the exact block of memory is known in advance by the compiler, and thus re-using the memory is cheap. This means that allocating on the stack has essentially no cost!

Arrays have to be heap allocated because their size (and thus the amount of memory they take up) is determined at runtime. But there are structures in Julia which are stack-allocated. `struct`s for example are stack-allocated "value-type"s. `Tuple`s are a stack-allocated collection. The most useful data structure for DiffEq though is the `StaticArray` from the package StaticArrays.jl. These arrays have their length determined at compile-time. They are created using macros attached to normal array expressions, for example:

```
using StaticArrays
A = @SVector [2.0,3.0,5.0]
```

```
3-element StaticArrays.SVector{3, Float64} with indices SOneTo(3):
 2.0
 3.0
 5.0
```

Notice that the `3` after `SVector` gives the size of the `SVector`. It cannot be changed. Additionally, `SVector`s are immutable, so we have to create a new `SVector` to change values. But remember, we don't have to worry about allocations because this data structure is stack-allocated. `SArray`s have a lot of extra optimizations as well: they have fast matrix multiplication, fast QR factorizations, etc. which directly make use of the information about the size of the array. Thus, when possible they should be used.

Unfortunately static arrays can only be used for sufficiently small arrays. After a certain size, they are forced to heap allocate after some instructions and their compile time balloons. Thus static arrays shouldn't be used if your system has more than 100 variables. Additionally, only the native Julia algorithms can fully utilize static arrays.

Let's ***optimize `lorenz` using static arrays***. Note that in this case, we want to use the out-of-place allocating form, but this time we want to output a static array:

```
function lorenz_static(u,p,t)
 dx = 10.0*(u[2]-u[1])
 dy = u[1]*(28.0-u[3]) - u[2]
 dz = u[1]*u[2] - (8/3)*u[3]
 @SVector [dx,dy,dz]
end
```

```
lorenz_static (generic function with 1 method)
```

To make the solver internally use static arrays, we simply give it a static array as the initial condition:

```
u0 = @SVector [1.0,0.0,0.0]
tspan = (0.0,100.0)
prob = ODEProblem(lorenz_static,u0,tspan)
@benchmark solve(prob,Tsit5())
```

```
BenchmarkTools.Trial:
  memory estimate:  446.73 KiB
  allocs estimate:  1314
  --------------
  minimum time:     310.586 μs (0.00% GC)
  median time:      324.706 μs (0.00% GC)
  mean time:        365.257 μs (6.47% GC)
  maximum time:     4.392 ms (89.20% GC)
  --------------
  samples:          10000
  evals/sample:     1
```

```julia
@benchmark solve(prob,Tsit5(),save_everystep=false)
```

```
BenchmarkTools.Trial:
  memory estimate:  3.69 KiB
  allocs estimate:  22
  --------------
  minimum time:     193.847 μs (0.00% GC)
  median time:      196.328 μs (0.00% GC)
  mean time:        198.075 μs (0.00% GC)
  maximum time:     463.604 μs (0.00% GC)
  --------------
  samples:          10000
  evals/sample:     1
```

And that's pretty much all there is to it. With static arrays you don't have to worry about allocating, so use operations like `*` and don't worry about fusing operations (discussed in the next section). Do "the vectorized code" of R/MATLAB/Python and your code in this case will be fast, or directly use the numbers/values.

Exercise 1 Implement the out-of-place array, in-place array, and out-of-place static array forms for the Henon-Heiles System and time the results.

0.2 Optimizing Large Systems

0.2.1 Interlude: Managing Allocations with Broadcast Fusion

When your system is sufficiently large, or you have to make use of a non-native Julia algorithm, you have to make use of `Arrays`. In order to use arrays in the most efficient manner, you need to be careful about temporary allocations. Vectorized calculations naturally have plenty of temporary array allocations. This is because a vectorized calculation outputs a vector. Thus:

```julia
A = rand(1000,1000); B = rand(1000,1000); C = rand(1000,1000)
test(A,B,C) = A + B + C
@benchmark test(A,B,C)
```

```
BenchmarkTools.Trial:
  memory estimate:  7.63 MiB
  allocs estimate:  2
  --------------
  minimum time:     1.117 ms (0.00% GC)
  median time:      1.203 ms (0.00% GC)
  mean time:        1.428 ms (15.74% GC)
```

```
  maximum time:     6.096 ms (75.78% GC)
  --------------
  samples:          3406
  evals/sample:     1
```

That expression `A + B + C` creates 2 arrays. It first creates one for the output of `A + B`, then uses that result array to `+ C` to get the final result. 2 arrays! We don't want that! The first thing to do to fix this is to use broadcast fusion. Broadcast fusion puts expressions together. For example, instead of doing the `+` operations separately, if we were to add them all at the same time, then we would only have a single array that's created. For example:

```
test2(A,B,C) = map((a,b,c)->a+b+c,A,B,C)
@benchmark test2(A,B,C)

BenchmarkTools.Trial:
  memory estimate:  7.63 MiB
  allocs estimate:  2
  --------------
  minimum time:     1.205 ms (0.00% GC)
  median time:      1.247 ms (0.00% GC)
  mean time:        1.479 ms (15.15% GC)
  maximum time:     4.299 ms (69.61% GC)
  --------------
  samples:          3292
  evals/sample:     1
```

Puts the whole expression into a single function call, and thus only one array is required to store output. This is the same as writing the loop:

```
function test3(A,B,C)
    D = similar(A)
    @inbounds for i in eachindex(A)
        D[i] = A[i] + B[i] + C[i]
    end
    D
end
@benchmark test3(A,B,C)

BenchmarkTools.Trial:
  memory estimate:  7.63 MiB
  allocs estimate:  2
  --------------
  minimum time:     1.120 ms (0.00% GC)
  median time:      1.196 ms (0.00% GC)
  mean time:        1.423 ms (15.74% GC)
  maximum time:     4.635 ms (73.09% GC)
  --------------
  samples:          3417
  evals/sample:     1
```

However, Julia's broadcast is syntactic sugar for this. If multiple expressions have a `.`, then it will put those vectorized operations together. Thus:

```
test4(A,B,C) = A .+ B .+ C
@benchmark test4(A,B,C)

BenchmarkTools.Trial:
  memory estimate:  7.63 MiB
  allocs estimate:  2
```

```
  --------------
  minimum time:     1.138 ms (0.00% GC)
  median time:      1.211 ms (0.00% GC)
  mean time:        1.436 ms (15.55% GC)
  maximum time:     4.399 ms (60.50% GC)
  --------------
  samples:          3387
  evals/sample:     1
```

is a version with only 1 array created (the output). Note that .s can be used with function calls as well:

```
sin.(A) .+ sin.(B)
```

```
1000×1000 Matrix{Float64}:
 1.27007   0.940957   0.292607  0.269155  …  1.29677   0.666315  0.840983
 0.554768  0.393315   0.747364  0.266694     0.586787  1.24692   1.02882
 0.4323    1.18211    0.387464  0.445846     0.674782  0.976628  1.05004
 0.831681  0.497684   1.02518   0.803792     0.878202  0.640039  0.864711
 0.274366  1.46029    1.52177   1.09442      0.348889  0.812259  0.997493
 0.707393  0.410531   0.912019  1.07618   …  0.860676  1.0568    1.08295
 1.22917   0.170894   0.55811   1.1302       0.893734  1.20009   0.853152
 1.33852   1.11614    0.715518  1.46507      1.42162   1.36284   1.16986
 1.1872    1.12002    0.778631  1.06046      1.27697   1.01936   1.33705
 0.954825  0.651031   0.469055  0.949534     1.06038   0.490776  0.54186
 ⋮                                        ⋱
 1.36709   0.0712647  0.287243  0.59944      1.03707   0.911209  0.486848
 0.947434  1.13506    1.49956   0.887114     0.649092  0.822763  0.349898
 1.4933    1.33322    0.171892  0.389535     1.25542   0.443911  0.599964
 0.84194   1.06661    1.47953   1.12068      0.419776  1.17319   0.875987
 0.791655  0.564508   0.889898  0.768667  …  1.02062   0.915082  0.667266
 0.369633  1.40935    1.17049   1.06198      1.1763    1.1813    1.21839
 1.34873   0.730056   0.43677   1.44254      0.824578  0.912898  1.04986
 1.32112   0.771721   1.24201   0.996659     0.920642  1.43637   1.37321
 0.19289   0.991586   0.989754  0.557391     0.510404  0.617162  1.04366
```

Also, the `@.` macro applys a dot to every operator:

```
test5(A,B,C) = @. A + B + C #only one array allocated
@benchmark test5(A,B,C)
```

```
BenchmarkTools.Trial:
  memory estimate:  7.63 MiB
  allocs estimate:  2
  --------------
  minimum time:     1.110 ms (0.00% GC)
  median time:      1.195 ms (0.00% GC)
  mean time:        1.420 ms (15.72% GC)
  maximum time:     4.458 ms (73.45% GC)
  --------------
  samples:          3425
  evals/sample:     1
```

Using these tools we can get rid of our intermediate array allocations for many vectorized function calls. But we are still allocating the output array. To get rid of that allocation, we can instead use mutation. Mutating broadcast is done via `.=`. For example, if we pre-allocate the output:

```
D = zeros(1000,1000);
```

Then we can keep re-using this cache for subsequent calculations. The mutating broadcasting form is:

```
test6!(D,A,B,C) = D .= A .+ B .+ C #only one array allocated
@benchmark test6!(D,A,B,C)
```

```
BenchmarkTools.Trial:
  memory estimate:  0 bytes
  allocs estimate:  0
  --------------
  minimum time:     1.114 ms (0.00% GC)
  median time:      1.128 ms (0.00% GC)
  mean time:        1.138 ms (0.00% GC)
  maximum time:     2.279 ms (0.00% GC)
  --------------
  samples:          4241
  evals/sample:     1
```

If we use `@.` before the `=`, then it will turn it into `.=`:

```
test7!(D,A,B,C) = @. D = A + B + C #only one array allocated
@benchmark test7!(D,A,B,C)
```

```
BenchmarkTools.Trial:
  memory estimate:  0 bytes
  allocs estimate:  0
  --------------
  minimum time:     1.099 ms (0.00% GC)
  median time:      1.131 ms (0.00% GC)
  mean time:        1.130 ms (0.00% GC)
  maximum time:     1.581 ms (0.00% GC)
  --------------
  samples:          4270
  evals/sample:     1
```

Notice that in this case, there is no "output", and instead the values inside of `D` are what are changed (like with the DiffEq inplace function). Many Julia functions have a mutating form which is denoted with a `!`. For example, the mutating form of the `map` is `map!`:

```
test8!(D,A,B,C) = map!((a,b,c)->a+b+c,D,A,B,C)
@benchmark test8!(D,A,B,C)
```

```
BenchmarkTools.Trial:
  memory estimate:  0 bytes
  allocs estimate:  0
  --------------
  minimum time:     2.329 ms (0.00% GC)
  median time:      2.365 ms (0.00% GC)
  mean time:        2.385 ms (0.00% GC)
  maximum time:     5.830 ms (0.00% GC)
  --------------
  samples:          2062
  evals/sample:     1
```

Some operations require using an alternate mutating form in order to be fast. For example, matrix multiplication via `*` allocates a temporary:

```
@benchmark A*B
```

```
BenchmarkTools.Trial:
  memory estimate:  7.63 MiB
  allocs estimate:  2
  --------------
  minimum time:     8.941 ms (0.00% GC)
  median time:      9.425 ms (0.00% GC)
  mean time:        9.743 ms (2.25% GC)
  maximum time:     17.534 ms (0.00% GC)
  --------------
  samples:          513
  evals/sample:     1
```

Instead, we can use the mutating form `mul!` into a cache array to avoid allocating the output:

```
using LinearAlgebra
@benchmark mul!(D,A,B) # same as D = A * B
```

```
BenchmarkTools.Trial:
  memory estimate:  0 bytes
  allocs estimate:  0
  --------------
  minimum time:     9.212 ms (0.00% GC)
  median time:      9.367 ms (0.00% GC)
  mean time:        9.474 ms (0.00% GC)
  maximum time:     17.307 ms (0.00% GC)
  --------------
  samples:          528
  evals/sample:     1
```

For repeated calculations this reduced allocation can stop GC cycles and thus lead to more efficient code. Additionally, ***we can fuse together higher level linear algebra operations using BLAS***. The package SugarBLAS.jl makes it easy to write higher level operations like `alpha*B*A + beta*C` as mutating BLAS calls.

0.2.2 Example Optimization: Gierer-Meinhardt Reaction-Diffusion PDE Discretization

Let's optimize the solution of a Reaction-Diffusion PDE's discretization. In its discretized form, this is the ODE:

$$du = D_1(A_y u + u A_x) + \frac{au^2}{v} + \bar{u} - \alpha u \tag{1}$$

$$dv = D_2(A_y v + v A_x) + au^2 + \beta v \tag{2}$$

where u, v, and A are matrices. Here, we will use the simplified version where A is the tridiagonal stencil $[1, -2, 1]$, i.e. it's the 2D discretization of the LaPlacian. The native code would be something along the lines of:

```
# Generate the constants
p = (1.0,1.0,1.0,10.0,0.001,100.0) # a,α,ubar,β,D1,D2
N = 100
Ax = Array(Tridiagonal([1.0 for i in 1:N-1],[-2.0 for i in 1:N],[1.0 for i in 1:N-1]))
Ay = copy(Ax)
Ax[2,1] = 2.0
```

```julia
Ax[end-1,end] = 2.0
Ay[1,2] = 2.0
Ay[end,end-1] = 2.0

function basic_version!(dr,r,p,t)
  a,α,ubar,β,D1,D2 = p
  u = r[:,:,1]
  v = r[:,:,2]
  Du = D1*(Ay*u + u*Ax)
  Dv = D2*(Ay*v + v*Ax)
  dr[:,:,1] = Du .+ a.*u.*u./v .+ ubar .- α*u
  dr[:,:,2] = Dv .+ a.*u.*u .- β*v
end

a,α,ubar,β,D1,D2 = p
uss = (ubar+β)/α
vss = (a/β)*uss^2
r0 = zeros(100,100,2)
r0[:,:,1] .= uss.+0.1.*rand.()
r0[:,:,2] .= vss

prob = ODEProblem(basic_version!,r0,(0.0,0.1),p)

ODEProblem with uType Array{Float64, 3} and tType Float64. In-place: true
timespan: (0.0, 0.1)
u0: 100×100×2 Array{Float64, 3}:
[:, :, 1] =
 11.0521  11.0566  11.0428  11.0379  …  11.0191  11.0977  11.0722  11.0083
 11.0501  11.0123  11.0165  11.0377     11.0559  11.0384  11.0594  11.0522
 11.029   11.0028  11.0099  11.089      11.0941  11.0243  11.0123  11.027
 11.0727  11.0049  11.0028  11.0479     11.0458  11.0627  11.0313  11.0088
 11.0183  11.0454  11.054   11.0384     11.0168  11.0088  11.0674  11.0448
 11.0444  11.003   11.059   11.0679  …  11.0028  11.0052  11.0968  11.0897
 11.0203  11.0844  11.0271  11.0649     11.0382  11.0436  11.0208  11.0396
 11.0222  11.0844  11.0209  11.0305     11.083   11.0008  11.093   11.0439
 11.0155  11.0985  11.0873  11.0865     11.0159  11.0673  11.0338  11.0484
 11.0998  11.049   11.0972  11.0044     11.0351  11.0008  11.0169  11.0097
  ⋮                                  ⋱
 11.0378  11.0439  11.0393  11.0775     11.0061  11.0722  11.0777  11.0116
 11.0395  11.0535  11.0709  11.0277     11.0358  11.0961  11.0953  11.0181
 11.0832  11.0368  11.0425  11.0834     11.025   11.029   11.0364  11.0605
 11.0326  11.0158  11.0193  11.0071     11.0748  11.0893  11.0705  11.0779
 11.0681  11.0269  11.0898  11.0962  …  11.0659  11.0805  11.0639  11.0492
 11.0845  11.0972  11.0955  11.062      11.0553  11.0462  11.0438  11.0768
 11.0055  11.0344  11.0954  11.0278     11.0065  11.0463  11.0931  11.0134
 11.0293  11.0057  11.099   11.0588     11.0939  11.078   11.0219  11.0794
 11.0766  11.0952  11.0815  11.0148     11.099   11.0949  11.0478  11.0096

[:, :, 2] =
 12.1  12.1  12.1  12.1  12.1  12.1  …  12.1  12.1  12.1  12.1  12.1  12.1
 12.1  12.1  12.1  12.1  12.1  12.1     12.1  12.1  12.1  12.1  12.1  12.1
 12.1  12.1  12.1  12.1  12.1  12.1     12.1  12.1  12.1  12.1  12.1  12.1
 12.1  12.1  12.1  12.1  12.1  12.1     12.1  12.1  12.1  12.1  12.1  12.1
 12.1  12.1  12.1  12.1  12.1  12.1     12.1  12.1  12.1  12.1  12.1  12.1
 12.1  12.1  12.1  12.1  12.1  12.1  …  12.1  12.1  12.1  12.1  12.1  12.1
 12.1  12.1  12.1  12.1  12.1  12.1     12.1  12.1  12.1  12.1  12.1  12.1
 12.1  12.1  12.1  12.1  12.1  12.1     12.1  12.1  12.1  12.1  12.1  12.1
 12.1  12.1  12.1  12.1  12.1  12.1     12.1  12.1  12.1  12.1  12.1  12.1
 12.1  12.1  12.1  12.1  12.1  12.1     12.1  12.1  12.1  12.1  12.1  12.1
```

```
  ⋮                                ⋮     ⋱         ⋮
 12.1  12.1  12.1  12.1  12.1  12.1     12.1  12.1  12.1  12.1  12.1  12.1
 12.1  12.1  12.1  12.1  12.1  12.1     12.1  12.1  12.1  12.1  12.1  12.1
 12.1  12.1  12.1  12.1  12.1  12.1     12.1  12.1  12.1  12.1  12.1  12.1
 12.1  12.1  12.1  12.1  12.1  12.1     12.1  12.1  12.1  12.1  12.1  12.1
 12.1  12.1  12.1  12.1  12.1  12.1  …   12.1  12.1  12.1  12.1  12.1  12.1
 12.1  12.1  12.1  12.1  12.1  12.1     12.1  12.1  12.1  12.1  12.1  12.1
 12.1  12.1  12.1  12.1  12.1  12.1     12.1  12.1  12.1  12.1  12.1  12.1
 12.1  12.1  12.1  12.1  12.1  12.1     12.1  12.1  12.1  12.1  12.1  12.1
 12.1  12.1  12.1  12.1  12.1  12.1     12.1  12.1  12.1  12.1  12.1  12.1
```

In this version we have encoded our initial condition to be a 3-dimensional array, with `u[:,:,1]` being the `A` part and `u[:,:,2]` being the `B` part.

```
@benchmark solve(prob,Tsit5())
```

```
BenchmarkTools.Trial:
  memory estimate:  194.54 MiB
  allocs estimate:  7647
  --------------
  minimum time:     70.364 ms (5.01% GC)
  median time:      75.320 ms (9.13% GC)
  mean time:        76.267 ms (8.55% GC)
  maximum time:     88.311 ms (4.09% GC)
  --------------
  samples:          66
  evals/sample:     1
```

While this version isn't very efficient,

We recommend writing the "high-level" code first, and iteratively optimizing it! The first thing that we can do is get rid of the slicing allocations. The operation `r[:,:,1]` creates a temporary array instead of a "view", i.e. a pointer to the already existing memory. To make it a view, add `@view`. Note that we have to be careful with views because they point to the same memory, and thus changing a view changes the original values:

```
A = rand(4)
@show A
B = @view A[1:3]
B[2] = 2
@show A
```

```
A = [0.31048903905426006, 0.017917973768470707, 0.6484323514489394, 0.63837
09466764833]
A = [0.31048903905426006, 2.0, 0.6484323514489394, 0.6383709466764833]
4-element Vector{Float64}:
 0.31048903905426006
 2.0
 0.6484323514489394
 0.6383709466764833
```

Notice that changing `B` changed `A`. This is something to be careful of, but at the same time we want to use this since we want to modify the output `dr`. Additionally, the last statement is a purely element-wise operation, and thus we can make use of broadcast fusion there. Let's rewrite `basic_version!` to ***avoid slicing allocations*** and to ***use broadcast fusion***:

```
function gm2!(dr,r,p,t)
  a,α,ubar,β,D1,D2 = p
  u = @view r[:,:,1]
  v = @view r[:,:,2]
  du = @view dr[:,:,1]
  dv = @view dr[:,:,2]
  Du = D1*(Ay*u + u*Ax)
  Dv = D2*(Ay*v + v*Ax)
  @. du = Du + a.*u.*u./v + ubar - α*u
  @. dv = Dv + a.*u.*u - β*v
end
prob = ODEProblem(gm2!,r0,(0.0,0.1),p)
@benchmark solve(prob,Tsit5())

BenchmarkTools.Trial:
  memory estimate:  124.66 MiB
  allocs estimate:  6117
  --------------
  minimum time:     57.254 ms (6.28% GC)
  median time:      62.295 ms (5.83% GC)
  mean time:        62.154 ms (7.22% GC)
  maximum time:     68.988 ms (9.82% GC)
  --------------
  samples:          81
  evals/sample:     1
```

Now, most of the allocations are taking place in `Du = D1*(Ay*u + u*Ax)` since those operations are vectorized and not mutating. We should instead replace the matrix multiplications with `mul!`. When doing so, we will need to have cache variables to write into. This looks like:

```
Ayu = zeros(N,N)
uAx = zeros(N,N)
Du = zeros(N,N)
Ayv = zeros(N,N)
vAx = zeros(N,N)
Dv = zeros(N,N)
function gm3!(dr,r,p,t)
  a,α,ubar,β,D1,D2 = p
  u = @view r[:,:,1]
  v = @view r[:,:,2]
  du = @view dr[:,:,1]
  dv = @view dr[:,:,2]
  mul!(Ayu,Ay,u)
  mul!(uAx,u,Ax)
  mul!(Ayv,Ay,v)
  mul!(vAx,v,Ax)
  @. Du = D1*(Ayu + uAx)
  @. Dv = D2*(Ayv + vAx)
  @. du = Du + a*u*u./v + ubar - α*u
  @. dv = Dv + a*u*u - β*v
end
prob = ODEProblem(gm3!,r0,(0.0,0.1),p)
@benchmark solve(prob,Tsit5())

BenchmarkTools.Trial:
  memory estimate:  31.22 MiB
  allocs estimate:  4893
  --------------
```

```
  minimum time:     54.989 ms (0.00% GC)
  median time:      59.105 ms (0.00% GC)
  mean time:        60.276 ms (1.84% GC)
  maximum time:     71.631 ms (4.74% GC)
  --------------
  samples:          83
  evals/sample:     1
```

But our temporary variables are global variables. We need to either declare the caches as `const` or localize them. We can localize them by adding them to the parameters, `p`. It's easier for the compiler to reason about local variables than global variables. ***Localizing variables helps to ensure type stability***.

```julia
p = (1.0,1.0,1.0,10.0,0.001,100.0,Ayu,uAx,Du,Ayv,vAx,Dv) # a,α,ubar,β,D1,D2
function gm4!(dr,r,p,t)
  a,α,ubar,β,D1,D2,Ayu,uAx,Du,Ayv,vAx,Dv = p
  u = @view r[:,:,1]
  v = @view r[:,:,2]
  du = @view dr[:,:,1]
  dv = @view dr[:,:,2]
  mul!(Ayu,Ay,u)
  mul!(uAx,u,Ax)
  mul!(Ayv,Ay,v)
  mul!(vAx,v,Ax)
  @. Du = D1*(Ayu + uAx)
  @. Dv = D2*(Ayv + vAx)
  @. du = Du + a*u*u./v + ubar - α*u
  @. dv = Dv + a*u*u - β*v
end
prob = ODEProblem(gm4!,r0,(0.0,0.1),p)
@benchmark solve(prob,Tsit5())

BenchmarkTools.Trial:
  memory estimate:  30.88 MiB
  allocs estimate:  1068
  --------------
  minimum time:     50.429 ms (0.00% GC)
  median time:      70.433 ms (0.00% GC)
  mean time:        72.447 ms (1.39% GC)
  maximum time:     87.979 ms (4.05% GC)
  --------------
  samples:          70
  evals/sample:     1
```

We could then use the BLAS `gemmv` to optimize the matrix multiplications some more, but instead let's devectorize the stencil.

```julia
p = (1.0,1.0,1.0,10.0,0.001,100.0,N)
function fast_gm!(du,u,p,t)
  a,α,ubar,β,D1,D2,N = p

  @inbounds for j in 2:N-1, i in 2:N-1
    du[i,j,1] = D1*(u[i-1,j,1] + u[i+1,j,1] + u[i,j+1,1] + u[i,j-1,1] - 4u[i,j,1]) +
              a*u[i,j,1]^2/u[i,j,2] + ubar - α*u[i,j,1]
  end

  @inbounds for j in 2:N-1, i in 2:N-1
    du[i,j,2] = D2*(u[i-1,j,2] + u[i+1,j,2] + u[i,j+1,2] + u[i,j-1,2] - 4u[i,j,2]) +
            a*u[i,j,1]^2 - β*u[i,j,2]
```

```julia
end

@inbounds for j in 2:N-1
  i = 1
  du[1,j,1] = D1*(2u[i+1,j,1] + u[i,j+1,1] + u[i,j-1,1] - 4u[i,j,1]) +
          a*u[i,j,1]^2/u[i,j,2] + ubar - α*u[i,j,1]
end
@inbounds for j in 2:N-1
  i = 1
  du[1,j,2] = D2*(2u[i+1,j,2] + u[i,j+1,2] + u[i,j-1,2] - 4u[i,j,2]) +
          a*u[i,j,1]^2 - β*u[i,j,2]
end
@inbounds for j in 2:N-1
  i = N
  du[end,j,1] = D1*(2u[i-1,j,1] + u[i,j+1,1] + u[i,j-1,1] - 4u[i,j,1]) +
         a*u[i,j,1]^2/u[i,j,2] + ubar - α*u[i,j,1]
end
@inbounds for j in 2:N-1
  i = N
  du[end,j,2] = D2*(2u[i-1,j,2] + u[i,j+1,2] + u[i,j-1,2] - 4u[i,j,2]) +
         a*u[i,j,1]^2 - β*u[i,j,2]
end

@inbounds for i in 2:N-1
  j = 1
  du[i,1,1] = D1*(u[i-1,j,1] + u[i+1,j,1] + 2u[i,j+1,1] - 4u[i,j,1]) +
            a*u[i,j,1]^2/u[i,j,2] + ubar - α*u[i,j,1]
end
@inbounds for i in 2:N-1
  j = 1
  du[i,1,2] = D2*(u[i-1,j,2] + u[i+1,j,2] + 2u[i,j+1,2] - 4u[i,j,2]) +
            a*u[i,j,1]^2 - β*u[i,j,2]
end
@inbounds for i in 2:N-1
  j = N
  du[i,end,1] = D1*(u[i-1,j,1] + u[i+1,j,1] + 2u[i,j-1,1] - 4u[i,j,1]) +
           a*u[i,j,1]^2/u[i,j,2] + ubar - α*u[i,j,1]
end
@inbounds for i in 2:N-1
  j = N
  du[i,end,2] = D2*(u[i-1,j,2] + u[i+1,j,2] + 2u[i,j-1,2] - 4u[i,j,2]) +
           a*u[i,j,1]^2 - β*u[i,j,2]
end

@inbounds begin
  i = 1; j = 1
  du[1,1,1] = D1*(2u[i+1,j,1] + 2u[i,j+1,1] - 4u[i,j,1]) +
            a*u[i,j,1]^2/u[i,j,2] + ubar - α*u[i,j,1]
  du[1,1,2] = D2*(2u[i+1,j,2] + 2u[i,j+1,2] - 4u[i,j,2]) +
            a*u[i,j,1]^2 - β*u[i,j,2]

  i = 1; j = N
  du[1,N,1] = D1*(2u[i+1,j,1] + 2u[i,j-1,1] - 4u[i,j,1]) +
           a*u[i,j,1]^2/u[i,j,2] + ubar - α*u[i,j,1]
  du[1,N,2] = D2*(2u[i+1,j,2] + 2u[i,j-1,2] - 4u[i,j,2]) +
           a*u[i,j,1]^2 - β*u[i,j,2]

  i = N; j = 1
  du[N,1,1] = D1*(2u[i-1,j,1] + 2u[i,j+1,1] - 4u[i,j,1]) +
```

```
            a*u[i,j,1]^2/u[i,j,2] + ubar - α*u[i,j,1]
    du[N,1,2] = D2*(2u[i-1,j,2] + 2u[i,j+1,2] - 4u[i,j,2]) +
            a*u[i,j,1]^2 - β*u[i,j,2]

    i = N; j = N
    du[end,end,1] = D1*(2u[i-1,j,1] + 2u[i,j-1,1] - 4u[i,j,1]) +
            a*u[i,j,1]^2/u[i,j,2] + ubar - α*u[i,j,1]
    du[end,end,2] = D2*(2u[i-1,j,2] + 2u[i,j-1,2] - 4u[i,j,2]) +
            a*u[i,j,1]^2 - β*u[i,j,2]
   end
end
prob = ODEProblem(fast_gm!,r0,(0.0,0.1),p)
@benchmark solve(prob,Tsit5())
```

```
BenchmarkTools.Trial:
  memory estimate:  30.85 MiB
  allocs estimate:  456
  --------------
  minimum time:     7.013 ms (0.00% GC)
  median time:      7.232 ms (0.00% GC)
  mean time:        8.206 ms (11.95% GC)
  maximum time:     12.137 ms (27.79% GC)
  --------------
  samples:          607
  evals/sample:     1
```

Lastly, we can do other things like multithread the main loops, but these optimizations get the last 2x-3x out. The main optimizations which apply everywhere are the ones we just performed (though the last one only works if your matrix is a stencil. This is known as a matrix-free implementation of the PDE discretization).

This gets us to about 8x faster than our original MATLAB/SciPy/R vectorized style code!

The last thing to do is then ***optimize our algorithm choice***. We have been using `Tsit5()` as our test algorithm, but in reality this problem is a stiff PDE discretization and thus one recommendation is to use `CVODE_BDF()`. However, instead of using the default dense Jacobian, we should make use of the sparse Jacobian afforded by the problem. The Jacobian is the matrix $\frac{df_i}{dr_j}$, where r is read by the linear index (i.e. down columns). But since the u variables depend on the v, the band size here is large, and thus this will not do well with a Banded Jacobian solver. Instead, we utilize sparse Jacobian algorithms. `CVODE_BDF` allows us to use a sparse Newton-Krylov solver by setting `linear_solver = :GMRES` (see the solver documentation, and thus we can solve this problem efficiently. Let's see how this scales as we increase the integration time.

```
prob = ODEProblem(fast_gm!,r0,(0.0,10.0),p)
@benchmark solve(prob,Tsit5())
```

```
BenchmarkTools.Trial:
  memory estimate:  2.76 GiB
  allocs estimate:  39336
  --------------
  minimum time:     1.051 s (38.57% GC)
  median time:      1.280 s (46.41% GC)
  mean time:        1.316 s (44.63% GC)
  maximum time:     1.653 s (47.32% GC)
  --------------
  samples:          4
  evals/sample:     1
```

```julia
using Sundials
@benchmark solve(prob,CVODE_BDF(linear_solver=:GMRES))
```

```
BenchmarkTools.Trial:
  memory estimate:  43.55 MiB
  allocs estimate:  6786
  --------------
  minimum time:     223.251 ms (0.00% GC)
  median time:      224.886 ms (0.00% GC)
  mean time:        225.631 ms (0.66% GC)
  maximum time:     228.017 ms (1.51% GC)
  --------------
  samples:          23
  evals/sample:     1
```

```julia
prob = ODEProblem(fast_gm!,r0,(0.0,100.0),p)
# Will go out of memory if we don't turn off `save_everystep`!
@benchmark solve(prob,Tsit5(),save_everystep=false)
```

```
BenchmarkTools.Trial:
  memory estimate:  2.91 MiB
  allocs estimate:  67
  --------------
  minimum time:     4.249 s (0.00% GC)
  median time:      4.296 s (0.00% GC)
  mean time:        4.296 s (0.00% GC)
  maximum time:     4.343 s (0.00% GC)
  --------------
  samples:          2
  evals/sample:     1
```

```julia
@benchmark solve(prob,CVODE_BDF(linear_solver=:GMRES))
```

```
BenchmarkTools.Trial:
  memory estimate:  242.29 MiB
  allocs estimate:  40965
  --------------
  minimum time:     1.387 s (0.00% GC)
  median time:      1.411 s (1.23% GC)
  mean time:        1.447 s (3.88% GC)
  maximum time:     1.578 s (12.03% GC)
  --------------
  samples:          4
  evals/sample:     1
```

Now let's check the allocation growth.

```julia
@benchmark solve(prob,CVODE_BDF(linear_solver=:GMRES),save_everystep=false)
```

```
BenchmarkTools.Trial:
  memory estimate:  2.88 MiB
  allocs estimate:  34149
  --------------
  minimum time:     1.359 s (0.00% GC)
  median time:      1.361 s (0.00% GC)
  mean time:        1.361 s (0.00% GC)
  maximum time:     1.363 s (0.00% GC)
  --------------
  samples:          4
  evals/sample:     1
```

```
prob = ODEProblem(fast_gm!,r0,(0.0,500.0),p)
@benchmark solve(prob,CVODE_BDF(linear_solver=:GMRES),save_everystep=false)
```

```
BenchmarkTools.Trial:
  memory estimate:  4.31 MiB
  allocs estimate:  58815
  --------------
  minimum time:     2.342 s (0.00% GC)
  median time:      2.344 s (0.00% GC)
  mean time:        2.343 s (0.00% GC)
  maximum time:     2.344 s (0.00% GC)
  --------------
  samples:          3
  evals/sample:     1
```

Notice that we've elimated almost all allocations, allowing the code to grow without hitting garbage collection and slowing down.

Why is `CVODE_BDF` doing well? What's happening is that, because the problem is stiff, the number of steps required by the explicit Runge-Kutta method grows rapidly, whereas `CVODE_BDF` is taking large steps. Additionally, the `GMRES` linear solver form is quite an efficient way to solve the implicit system in this case. This is problem-dependent, and in many cases using a Krylov method effectively requires a preconditioner, so you need to play around with testing other algorithms and linear solvers to find out what works best with your problem.

0.3 Conclusion

Julia gives you the tools to optimize the solver "all the way", but you need to make use of it. The main thing to avoid is temporary allocations. For small systems, this is effectively done via static arrays. For large systems, this is done via in-place operations and cache arrays. Either way, the resulting solution can be immensely sped up over vectorized formulations by using these principles.

0.4 Appendix

These tutorials are a part of the SciMLTutorials.jl repository, found at: https://github.com/SciML/SciML For more information on high-performance scientific machine learning, check out the SciML Open Source Software Organization https://sciml.ai.

To locally run this tutorial, do the following commands:

```
using SciMLTutorials
SciMLTutorials.weave_file("tutorials/introduction","03-optimizing_diffeq_code.jmd")
```

Computer Information:

```
Julia Version 1.6.1
Commit 6aaedecc44 (2021-04-23 05:59 UTC)
Platform Info:
  OS: Linux (x86_64-pc-linux-gnu)
```

```
  CPU: AMD EPYC 7502 32-Core Processor
  WORD_SIZE: 64
  LIBM: libopenlibm
  LLVM: libLLVM-11.0.1 (ORCJIT, znver2)
Environment:
  JULIA_DEPOT_PATH = /root/.cache/julia-buildkite-plugin/depots/a6029d3a-f78b-41ea-bc9
  JULIA_NUM_THREADS = 16
```

Package Information:

```
      Status `/var/lib/buildkite-agent/builds/6-amdci4-julia-csail-mit-edu/julialang/s
  [6e4b80f9] BenchmarkTools v1.0.0
  [0c46a032] DifferentialEquations v6.17.1
  [65888b18] ParameterizedFunctions v5.10.0
  [91a5bcdd] Plots v1.15.2
  [30cb0354] SciMLTutorials v0.9.0
  [90137ffa] StaticArrays v1.2.0
  [c3572dad] Sundials v4.4.3
  [37e2e46d] LinearAlgebra
```

And the full manifest:

```
      Status `/var/lib/buildkite-agent/builds/6-amdci4-julia-csail-mit-edu/julialang/s
  [c3fe647b] AbstractAlgebra v0.16.0
  [1520ce14] AbstractTrees v0.3.4
  [79e6a3ab] Adapt v3.3.0
  [ec485272] ArnoldiMethod v0.1.0
  [4fba245c] ArrayInterface v3.1.15
  [4c555306] ArrayLayouts v0.7.0
  [aae01518] BandedMatrices v0.16.9
  [6e4b80f9] BenchmarkTools v1.0.0
  [764a87c0] BoundaryValueDiffEq v2.7.1
  [fa961155] CEnum v0.4.1
  [d360d2e6] ChainRulesCore v0.9.44
  [b630d9fa] CheapThreads v0.2.5
  [35d6a980] ColorSchemes v3.12.1
  [3da002f7] ColorTypes v0.11.0
  [5ae59095] Colors v0.12.8
  [861a8166] Combinatorics v1.0.2
  [38540f10] CommonSolve v0.2.0
  [bbf7d656] CommonSubexpressions v0.3.0
  [34da2185] Compat v3.30.0
  [8f4d0f93] Conda v1.5.2
  [187b0558] ConstructionBase v1.2.1
  [d38c429a] Contour v0.5.7
  [9a962f9c] DataAPI v1.6.0
  [864edb3b] DataStructures v0.18.9
```

```
[e2d170a0] DataValueInterfaces v1.0.0
[bcd4f6db] DelayDiffEq v5.31.0
[2b5f629d] DiffEqBase v6.62.2
[459566f4] DiffEqCallbacks v2.16.1
[5a0ffddc] DiffEqFinancial v2.4.0
[c894b116] DiffEqJump v6.14.2
[77a26b50] DiffEqNoiseProcess v5.7.3
[055956cb] DiffEqPhysics v3.9.0
[163ba53b] DiffResults v1.0.3
[b552c78f] DiffRules v1.0.2
[0c46a032] DifferentialEquations v6.17.1
[c619ae07] DimensionalPlotRecipes v1.2.0
[b4f34e82] Distances v0.10.3
[31c24e10] Distributions v0.24.18
[ffbed154] DocStringExtensions v0.8.4
[d4d017d3] ExponentialUtilities v1.8.4
[e2ba6199] ExprTools v0.1.3
[c87230d0] FFMPEG v0.4.0
[7034ab61] FastBroadcast v0.1.8
[9aa1b823] FastClosures v0.3.2
[1a297f60] FillArrays v0.11.7
[6a86dc24] FiniteDiff v2.8.0
[53c48c17] FixedPointNumbers v0.8.4
[59287772] Formatting v0.4.2
[f6369f11] ForwardDiff v0.10.18
[069b7b12] FunctionWrappers v1.1.2
[28b8d3ca] GR v0.57.4
[5c1252a2] GeometryBasics v0.3.12
[42e2da0e] Grisu v1.0.2
[cd3eb016] HTTP v0.9.9
[eafb193a] Highlights v0.4.5
[0e44f5e4] Hwloc v2.0.0
[7073ff75] IJulia v1.23.2
[615f187c] IfElse v0.1.0
[d25df0c9] Inflate v0.1.2
[83e8ac13] IniFile v0.5.0
[c8e1da08] IterTools v1.3.0
[42fd0dbc] IterativeSolvers v0.9.1
[82899510] IteratorInterfaceExtensions v1.0.0
[692b3bcd] JLLWrappers v1.3.0
[682c06a0] JSON v0.21.1
[b964fa9f] LaTeXStrings v1.2.1
[2ee39098] LabelledArrays v1.6.1
[23fbe1c1] Latexify v0.15.5
[093fc24a] LightGraphs v1.3.5
[d3d80556] LineSearches v7.1.1
[2ab3a3ac] LogExpFunctions v0.2.4
[bdcacae8] LoopVectorization v0.12.23
[1914dd2f] MacroTools v0.5.6
```

```
[739be429] MbedTLS v1.0.3
[442fdcdd] Measures v0.3.1
[e1d29d7a] Missings v1.0.0
[961ee093] ModelingToolkit v5.16.0
[46d2c3a1] MuladdMacro v0.2.2
[f9640e96] MultiScaleArrays v1.8.1
[ffc61752] Mustache v1.0.10
[d41bc354] NLSolversBase v7.8.0
[2774e3e8] NLsolve v4.5.1
[77ba4419] NaNMath v0.3.5
[8913a72c] NonlinearSolve v0.3.8
[6fe1bfb0] OffsetArrays v1.9.0
[429524aa] Optim v1.3.0
[bac558e1] OrderedCollections v1.4.1
[1dea7af3] OrdinaryDiffEq v5.56.0
[90014a1f] PDMats v0.11.0
[65888b18] ParameterizedFunctions v5.10.0
[d96e819e] Parameters v0.12.2
[69de0a69] Parsers v1.1.0
[ccf2f8ad] PlotThemes v2.0.1
[995b91a9] PlotUtils v1.0.10
[91a5bcdd] Plots v1.15.2
[e409e4f3] PoissonRandom v0.4.0
[f517fe37] Polyester v0.3.1
[85a6dd25] PositiveFactorizations v0.2.4
[21216c6a] Preferences v1.2.2
[1fd47b50] QuadGK v2.4.1
[74087812] Random123 v1.3.1
[fb686558] RandomExtensions v0.4.3
[e6cf234a] RandomNumbers v1.4.0
[3cdcf5f2] RecipesBase v1.1.1
[01d81517] RecipesPipeline v0.3.2
[731186ca] RecursiveArrayTools v2.11.4
[f2c3362d] RecursiveFactorization v0.1.12
[189a3867] Reexport v1.0.0
[ae029012] Requires v1.1.3
[ae5879a3] ResettableStacks v1.1.0
[79098fc4] Rmath v0.7.0
[7e49a35a] RuntimeGeneratedFunctions v0.5.2
[476501e8] SLEEFPirates v0.6.20
[1bc83da4] SafeTestsets v0.0.1
[0bca4576] SciMLBase v1.13.4
[30cb0354] SciMLTutorials v0.9.0
[6c6a2e73] Scratch v1.0.3
[efcf1570] Setfield v0.7.0
[992d4aef] Showoff v1.0.3
[699a6c99] SimpleTraits v0.9.3
[b85f4697] SoftGlobalScope v1.1.0
[a2af1166] SortingAlgorithms v1.0.0
```

```
[47a9eef4] SparseDiffTools v1.13.2
[276daf66] SpecialFunctions v1.4.1
[aedffcd0] Static v0.2.4
[90137ffa] StaticArrays v1.2.0
[82ae8749] StatsAPI v1.0.0
[2913bbd2] StatsBase v0.33.8
[4c63d2b9] StatsFuns v0.9.8
[9672c7b4] SteadyStateDiffEq v1.6.2
[789caeaf] StochasticDiffEq v6.34.1
[7792a7ef] StrideArraysCore v0.1.11
[09ab397b] StructArrays v0.5.1
[c3572dad] Sundials v4.4.3
[d1185830] SymbolicUtils v0.11.2
[0c5d862f] Symbolics v0.1.25
[3783bdb8] TableTraits v1.0.1
[bd369af6] Tables v1.4.2
[8290d209] ThreadingUtilities v0.4.4
[a759f4b9] TimerOutputs v0.5.9
[a2a6695c] TreeViews v0.3.0
[5c2747f8] URIs v1.3.0
[3a884ed6] UnPack v1.0.2
[1986cc42] Unitful v1.7.0
[3d5dd08c] VectorizationBase v0.20.11
[81def892] VersionParsing v1.2.0
[19fa3120] VertexSafeGraphs v0.1.2
[44d3d7a6] Weave v0.10.8
[ddb6d928] YAML v0.4.6
[c2297ded] ZMQ v1.2.1
[700de1a5] ZygoteRules v0.2.1
[6e34b625] Bzip2_jll v1.0.6+5
[83423d85] Cairo_jll v1.16.0+6
[5ae413db] EarCut_jll v2.1.5+1
[2e619515] Expat_jll v2.2.10+0
[b22a6f82] FFMPEG_jll v4.3.1+4
[a3f928ae] Fontconfig_jll v2.13.1+14
[d7e528f0] FreeType2_jll v2.10.1+5
[559328eb] FriBidi_jll v1.0.5+6
[0656b61e] GLFW_jll v3.3.4+0
[d2c73de3] GR_jll v0.57.2+0
[78b55507] Gettext_jll v0.21.0+0
[7746bdde] Glib_jll v2.68.1+0
[e33a78d0] Hwloc_jll v2.4.1+0
[aacddb02] JpegTurbo_jll v2.0.1+3
[c1c5ebd0] LAME_jll v3.100.0+3
[dd4b983a] LZO_jll v2.10.1+0
[dd192d2f] LibVPX_jll v1.9.0+1
[e9f186c6] Libffi_jll v3.2.2+0
[d4300ac3] Libgcrypt_jll v1.8.7+0
[7e76a0d4] Libglvnd_jll v1.3.0+3
```

```
[7add5ba3] Libgpg_error_jll v1.42.0+0
[94ce4f54] Libiconv_jll v1.16.1+0
[4b2f31a3] Libmount_jll v2.35.0+0
[89763e89] Libtiff_jll v4.1.0+2
[38a345b3] Libuuid_jll v2.36.0+0
[e7412a2a] Ogg_jll v1.3.4+2
[458c3c95] OpenSSL_jll v1.1.1+6
[efe28fd5] OpenSpecFun_jll v0.5.4+0
[91d4177d] Opus_jll v1.3.1+3
[2f80f16e] PCRE_jll v8.44.0+0
[30392449] Pixman_jll v0.40.1+0
[ea2cea3b] Qt5Base_jll v5.15.2+0
[f50d1b31] Rmath_jll v0.3.0+0
[fb77eaff] Sundials_jll v5.2.0+1
[a2964d1f] Wayland_jll v1.17.0+4
[2381bf8a] Wayland_protocols_jll v1.18.0+4
[02c8fc9c] XML2_jll v2.9.12+0
[aed1982a] XSLT_jll v1.1.34+0
[4f6342f7] Xorg_libX11_jll v1.6.9+4
[0c0b7dd1] Xorg_libXau_jll v1.0.9+4
[935fb764] Xorg_libXcursor_jll v1.2.0+4
[a3789734] Xorg_libXdmcp_jll v1.1.3+4
[1082639a] Xorg_libXext_jll v1.3.4+4
[d091e8ba] Xorg_libXfixes_jll v5.0.3+4
[a51aa0fd] Xorg_libXi_jll v1.7.10+4
[d1454406] Xorg_libXinerama_jll v1.1.4+4
[ec84b674] Xorg_libXrandr_jll v1.5.2+4
[ea2f1a96] Xorg_libXrender_jll v0.9.10+4
[14d82f49] Xorg_libpthread_stubs_jll v0.1.0+3
[c7cfdc94] Xorg_libxcb_jll v1.13.0+3
[cc61e674] Xorg_libxkbfile_jll v1.1.0+4
[12413925] Xorg_xcb_util_image_jll v0.4.0+1
[2def613f] Xorg_xcb_util_jll v0.4.0+1
[975044d2] Xorg_xcb_util_keysyms_jll v0.4.0+1
[0d47668e] Xorg_xcb_util_renderutil_jll v0.3.9+1
[c22f9ab0] Xorg_xcb_util_wm_jll v0.4.1+1
[35661453] Xorg_xkbcomp_jll v1.4.2+4
[33bec58e] Xorg_xkeyboard_config_jll v2.27.0+4
[c5fb5394] Xorg_xtrans_jll v1.4.0+3
[8f1865be] ZeroMQ_jll v4.3.2+6
[3161d3a3] Zstd_jll v1.5.0+0
[0ac62f75] libass_jll v0.14.0+4
[f638f0a6] libfdk_aac_jll v0.1.6+4
[b53b4c65] libpng_jll v1.6.38+0
[a9144af2] libsodium_jll v1.0.20+0
[f27f6e37] libvorbis_jll v1.3.6+6
[1270edf5] x264_jll v2020.7.14+2
[dfaa095f] x265_jll v3.0.0+3
[d8fb68d0] xkbcommon_jll v0.9.1+5
```

```
[0dad84c5] ArgTools
[56f22d72] Artifacts
[2a0f44e3] Base64
[ade2ca70] Dates
[8bb1440f] DelimitedFiles
[8ba89e20] Distributed
[f43a241f] Downloads
[7b1f6079] FileWatching
[9fa8497b] Future
[b77e0a4c] InteractiveUtils
[b27032c2] LibCURL
[76f85450] LibGit2
[8f399da3] Libdl
[37e2e46d] LinearAlgebra
[56ddb016] Logging
[d6f4376e] Markdown
[a63ad114] Mmap
[ca575930] NetworkOptions
[44cfe95a] Pkg
[de0858da] Printf
[3fa0cd96] REPL
[9a3f8284] Random
[ea8e919c] SHA
[9e88b42a] Serialization
[1a1011a3] SharedArrays
[6462fe0b] Sockets
[2f01184e] SparseArrays
[10745b16] Statistics
[4607b0f0] SuiteSparse
[fa267f1f] TOML
[a4e569a6] Tar
[8dfed614] Test
[cf7118a7] UUIDs
[4ec0a83e] Unicode
[e66e0078] CompilerSupportLibraries_jll
[deac9b47] LibCURL_jll
[29816b5a] LibSSH2_jll
[c8ffd9c3] MbedTLS_jll
[14a3606d] MozillaCACerts_jll
[4536629a] OpenBLAS_jll
[bea87d4a] SuiteSparse_jll
[83775a58] Zlib_jll
[8e850ede] nghttp2_jll
[3f19e933] p7zip_jll
```