Notes 5: Efficiency

Chris Paciorek

2025-02-04

Table of contents

Introduction

This document is the fifth of a set of notes, this document focusing on writing efficient Julia code. The notes are not meant to be particularly complete in terms of useful functions (Google and LLMs can now provide that quite well), but rather to introduce the language and consider key programming concepts in the context of Julia.

Given that, the document heavily relies on demos, with interpretation in some cases left to the reader.

Timing

Being able to time code is critical for understanding and improving efficiency.

> ⚠ Compilation time
>
> With Julia, we need to pay particular attention to the effect of just-in-time (JIT) compilation on timing. The first time a function is called with specific set of argument types, Julia will compile the method that is invoked. We generally don't want to time the compilation, only the run time, assuming the function will be run repeatedly with a given set of argument types.

`@time` is a macro that will time some code. However, it's better to use `@btime` from `BenchmarkTools` as that will run the code multiple times and will make sure not to count the compilation time.

```
function myexp!(x)
  for i in 1:length(x)
    x[i] = exp(x[i])
  end
end

n = Int(1e7)
y = rand(n);
@time myexp!(y) ## Compilation time included.
```

```
  0.075221 seconds (2.62 k allocations: 178.000 KiB, 10.40% compilation time)
```

```
y = rand(n);
@time myexp!(y) ## Compilation time not included.
```

```
  0.069284 seconds
```

```
using BenchmarkTools

y = rand(n);
@btime myexp!(y)
```

```
  59.500 ms (0 allocations: 0 bytes)
```

> ⚠ Exercise
>
> How long does that loop take in R or Python? What about a vectorized solution in R or Python?

We can time a block of code, but I'm not sure what Julia does in terms of JIT for code that is not in functions. You may discover more in working on the fourth problem of PS2.

```
@btime begin
y = 3
z = 7
end
```

```
  1.299 ns (0 allocations: 0 bytes)
```

```
7
```

Profiling

Profiling involves timing each step in a set of code. One can use the Profile module to do this in Julia.

One thing to keep in mind when profiling is whether the timing for nested function calls is included in the timing of the function that makes the nested function calls.

```
using Profile

function ols_slow(y::Vector{<:Number}, X::Matrix{<:Number})
    xtx = X'X;
    xty = X'y;
    xtxinverse = inv(xtx);  ## This is an inefficient approach.
    return xtxinverse * xty
end

n = Int(1e4)
p = 2000
y = randn(n);
X = randn((n,p));

## Run once to avoid profiling JIT compilation.
coefs = ols_slow(y, X);
```

Directly interpreting the Profile output can be difficulty. In this case, if we ran the following code, we'd see very long, hard-to-interpret information.

```
@profile coefs = ols_slow(y, X)
Profile.print()
```

Instead let's try a visualization. There are other Julia packages for visualizing profiler output. Some might be better than this. (I tried `ProfileView` and liked `StatProfilerHTML` better.)

```
using ProfileView
@profview ols_slow(y, X)

using StatProfilerHTML
@profilehtml ols_slow(y, X)
```

`@profilehtml produces [this output](statprof/index.html), which can in some ways be hard to interpret, but the color-coded division between`inv,*and*` gives us an idea of where time is being spent. That output might not show up fully in the links - you might need to run the code above yourself.

Pre-allocation

In R (also with numpy arrays in Python), it's a bad idea to iteratively increase the size of an object, such as doing this:

```
n <- 5000
x <- 1
for(i in 2:n)
  x <- c(x, i)
```

Python lists handle this much better by allocating increasingly large additional amounts of memory as the object grows when using `.append()`.

Let's consider this in Julia.

```
function fun_prealloc(n)
  x = zeros(n);
  for i in 1:n
    x[i] = i;
  end
  return x
end

function fun_grow(n)
  x = Float64[];
  for i in 1:n
    push!(x, i);
  end
  return x
end

using BenchmarkTools

n = 100000000
@btime x1 = fun_prealloc(n);
```

```
  331.161 ms (2 allocations: 762.94 MiB)
```

```
@btime x2 = fun_grow(n);
```

```
  1.679 s (23 allocations: 1019.60 MiB)
```

That indicates that it's better to pre-allocate memory in Julia, but the time does not seem to grow as order of n^2 as it does in R or with numpy arrays. So that suggests Julia is growing the array in a smart fashion.

We can verify that by looking at the memory allocation information returned by `@btime`.

For `fun_prealloc`, we see an allocation of ~800 MB, consistent with allocating an array of 100 million 8 byte floats. (It turns out the "second" allocation occurs because we are running `@btime` in the global scope).

For `fun_grow`, we see 23 allocations of ~1 GB, consistent with Julia growing the array in a smart fashion but with some additional memory allocation.

If the array were reallocated each time it grew by one, we'd allocate and copy $1+2+\cdots+n = n(n+1)/2$ numbers in total over the course of the computation (but not all at once), which would take a lot of time.

Vectorization

As we've seen, the vectorized versions of functions have a dot after the function name (or before an operator).

```
x = ["spam", 2.0, 5, [10, 20]]
length(x)
```

```
4
```

```
length.(x)
```

```
4-element Vector{Int64}:
 4
 1
 1
 2
```

```
map(length, x)
```

```
4-element Vector{Int64}:
 4
 1
 1
 2
```

```
x = [2.1, 3.1, 5.3, 7.9]
x .+ 10
```

```
4-element Vector{Float64}:
 12.1
 13.1
 15.3
 17.9
```

```
x + x
```

```
4-element Vector{Float64}:
  4.2
  6.2
 10.6
 15.8
```

```
x .> 5.0
```

```
4-element BitVector:
 0
 0
 1
 1
```

```
x .== 3.1
```

```
4-element BitVector:
 0
 1
 0
 0
```

Unlike in Python or R, it shouldn't matter for efficiency if you use a vectorized function or write a loop, because with Julia's just-in-time compilation, the compiled code should be similar. (This assumes your code is inside a function.) So the main appeal of vectorization is code clarity and ease of writing the code.

We can automatically use the dot vectorization with functions we write:

```
function plus3(x)
  return x + 3
end

plus3.(x)
```

```
4-element Vector{Float64}:
  5.1
  6.1
  8.3
 10.9
```

This invokes `broadcast(plus3, args...)`.

Broadcasting will happen over multiple arguments if more than one argument is an array.

Consider the difference between the following vectorized calls:

```
x = randn(5)
 = 10;
y1 = x .+   .* randn.()
y2 = x .+   .* randn()
print((y1 - x) /  )
print((y2 - x) /  )
```

```
[0.4170656514220214, -0.8850493432768529, 0.1260295830134405, -0.22061211497134475, -0.19497228825436
```

That's perhaps a bit surprising given one might think that because the multiplication is done first, the `.* randn.()` might produce a scalar, as it does if you just run `.* randn.()` on its own.

Loop fusion

If one runs a vectorized calculation that involves multiple steps in a language like R or Python, there are some inefficiencies.

Consider this computation:

```
x = tan(x) + 3*sin(x)
```

If run as vectorized code in a language like R or Python, it's much faster than using a loop, but it does have some downsides.

- First, it will use additional memory (temporary arrays will be created to store `tan(x)`, `sin(x)`, `3*sin(x)`). (We can consider what the abstract syntax tree would be for that calculation.)
- Second, multiple for loops will have to get executed when the vectorized code is run, looping over the elements of `x` to calculate `tan(x)`, `sin(x)`, etc. (For example in R or Python/numpy, multiple for loops would get run in the underlying C code.)

In contrast, running via a for loop (in R or Python or Julia) avoids the temporary arrays and involves a single loop:

```
for i in 1:length(x)
    x[i] = tan(x[i]) + 3*sin(x[i])
end
```

Thankfully, Julia "fuses" the loops of vectorized code automatically when one uses the dot syntax for vectorization, so one shouldn't suffer from the downsides of vectorization. One could of course use a loop in Julia, and it should be fast, but it's more code to write and harder to read.

Memory allocation with loop fusion

Let's look at memory allocation when putting the code into a function:

```
function mymath(x)
   return tan(x) + 3*sin(x)
end

function mymathloop(x)
  for i in 1:length(x)
    x[i] = tan(x[i]) + 3*sin(x[i])
  end
  return x
end

n = 100000000;
x = rand(n);

@btime y = mymath.(x);
```

```
  2.525 s (3 allocations: 762.94 MiB)
```

```
@btime y = mymathloop(x);
```

```
  3.066 s (0 allocations: 0 bytes)
```

Note that it appears only 800 MB (~760 MiB; ~0.95 MiB = 1 MB) are allocated (for the output) in the (presumably) fused operation, rather than multiples of 800 MB for various temporary arrays that one might expect to be created.

And in the loop, there is no allocation. We might expect some allocation of scalars, but those are probably handled differently than allocating memory for arrays off the heap. I've seen some information for how Julia handles allocation of space for immutable objects (including scalars and strings), but I hvaen't had a chance to absorb that.

Cases without loop fusion

We can do addition or subtraction of two arrays or multiplication/division with array and scalar without the "dot" vectorization. However, as seen with the additional memory allocation here, the loop fusion is not done.

```
function mymath2(x)
   return 3*x+x/7
end

@btime y = mymath2(x);
```

```
  1.065 s (6 allocations: 2.24 GiB)
```

In contrast, here we see only the allocation for the output object.

```
@btime y = mymath2.(x);
```

```
  454.403 ms (3 allocations: 762.94 MiB)
```

Cache-aware programming and array storage

Julia stores the values in a matrix contiguously column by column (and analogously for higher-dimensional arrays).

We should therefore access matrix elements within a column rather than within a row. Why is that?

> **Memory access and the cache**
>
> When a value is retrieved from main memory into the CPU cache, a block of values will be retrieved, and those will generally include the values in the same column but (for large enough arrays) not all the values in the same row. If subsequent operations work on values from that column, the values won't need to be moved into the cache. (This is called a "cache hit").

Let's first see if it makes a difference when using Julia's built-in `sum` function, which can do the reduction operation on various dimensions of the array.

```
using Random
using BenchmarkTools

nr = 800000;
nc = 100;
A = randn(nr, nc);    # long matrix
tA = randn(nc, nr);   # wide matrix
```

```julia
function sum_by_column(X)
    return sum(X, dims=1)
end

function sum_by_row(X)
    return sum(X, dims=2)
end

@btime tmp = sum_by_column(A);
```

```
  39.221 ms (1 allocation: 896 bytes)
```

```julia
@btime tmp = sum_by_row(tA);
```

```
  43.396 ms (5 allocations: 976 bytes)
```

There's little difference.

Are we wrong about how the cache works? Probably not; rather it's probably that Julia's `sum()` is set up to take advantage of how the cache works by being careful about the order of operations used to sum the rows or columns.

> **Exercise**
>
> How could you program the for loops involved in row-wise summation to be efficient when a matrix is stored column-major given how caching work? If you retrieve the data by column, how do you get the row sums?

In contrast, if we manually loop over rows or columns, we do see a big (almost order-of-magnitude) difference.

```julia
@btime tmp = [sum(A[:,col]) for col in 1:size(A,2)];
```

```
  135.859 ms (405 allocations: 610.36 MiB)
```

```julia
@btime tmp = [sum(A[row,:]) for row in 1:size(A,1)];
```

```
  778.063 ms (4798474 allocations: 750.71 MiB)
```

So while one lesson is to code with the cache in mind, another is to use built-in functions that are probably written for efficiency.

> **Exercise**
>
> In your own work, can you think of an algorithm and associated data structures where one has to retrieve a lot of data and one would want to think about cache hits and misses? In general the idea is that if you retrieve a value, try to make use of the nearby values at that same time, rather than retrieving the nearby values later on in the computation.

Store values contiguously in memory

If we are storing an array of all the same type of values, these can be stored contiguously. That's not the case with abstract types.

For example, here `Real` values can vary in size.

```
a = Real[]
sizeof(a)
push!(a, 3.5)
sizeof(a)
push!(a, Int16(2))
sizeof(a[2])
sizeof(a)
```

```
16
```

And we see that having an array of Reals is bad for performance. As part of this notice the additional allocation.

```
using LinearAlgebra
n = 100;
A = rand(n, n);
@btime tmp = A'A;  # Equivalent to A' * A or transpose(A) * A.
```

```
  45.131  s (3 allocations: 78.19 KiB)
```

```
100×100 Matrix{Float64}:
 31.3434  28.2991  23.3785  22.2482  …  22.8816  23.144   25.6776  23.1401
 28.2991  40.381   26.3591  28.3626     25.8735  27.3842  29.9671  26.2799
 23.3785  26.3591  32.6379  23.8486     25.1776  23.2087  27.3518  23.6795
 22.2482  28.3626  23.8486  31.2422     22.1873  23.8796  25.6417  22.2948
 24.3474  29.1835  26.0694  26.0089     25.9571  28.0218  28.0398  23.3216
 24.2326  28.4558  26.0709  25.0149  …  25.407   27.7716  28.9334  24.154
 24.4235  27.5001  24.3027  24.3611     24.7207  25.1982  26.7347  25.5547
 22.6136  26.6354  25.0601  23.7247     26.0032  22.8209  25.973   23.3979
 25.0136  28.6411  23.6948  23.5412     24.0284  24.2705  28.0546  25.2663
 22.4926  26.0361  25.136   23.4652     23.6698  24.1082  27.5221  21.1363
 21.0514  25.0219  21.9994  22.6873  …  22.9995  23.0049  25.1662  23.2241
 24.9928  27.1519  24.073   23.8966     23.7997  24.0047  27.19    23.5732
 27.7282  28.9997  27.4725  25.8123     26.6285  26.9111  28.3441  25.1172
  ⋮                                  ⋱
 23.3272  27.7131  24.4456  22.9734     22.1652  25.582   26.7593  22.6326
 23.9461  29.2237  25.4995  25.3786     24.2368  27.2959  27.5395  25.5569
 23.9127  27.8013  26.6871  24.2284  …  26.1844  27.3045  28.2586  26.2357
 25.0488  27.6548  24.1603  22.9998     25.6098  24.2775  26.4557  23.6312
 25.884   30.9974  27.4021  26.2885     26.5123  27.2523  29.6701  26.254
 25.3851  27.0305  25.8697  23.3191     25.1691  24.5485  27.5205  25.2234
 24.5266  29.1042  26.2418  25.2955     23.7441  26.1753  26.3628  24.2068
 23.3792  27.1223  24.3354  23.7604  …  22.5078  26.2251  27.2906  23.7979
```

```
 22.8816  25.8735  25.1776  22.1873     32.8427  22.7119  26.4834  22.0585
 23.144   27.3842  23.2087  23.8796     22.7119  32.9915  25.9671  23.43
 25.6776  29.9671  27.3518  25.6417     26.4834  25.9671  36.366   24.7744
 23.1401  26.2799  23.6795  22.2948     22.0585  23.43    24.7744  29.8628
```

```
rA = convert(Array{Real}, A);
@btime tmp = rA'rA;
```

```
  41.008 ms (2030004 allocations: 31.05 MiB)
```

Lookup speed

If we have code that needs to retrieve a lot of values from a data structure, it's worth knowing the situations in which we can expect that lookup to be fast.

Lookup in arrays is fast ($O(1)$; i.e., not varying with the size of the array) because of the "random access" aspect of RAM (random access memory).

```
n=Int(1e7);

x = randn(n);
ind = Int(n/2);
@btime x[ind];
```

```
  19.613 ns (1 allocation: 16 bytes)
```

```
y = rand(10);
@btime y[5];
```

```
  19.994 ns (1 allocation: 16 bytes)
```

Next, lookup in a Julia dictionary is fast $O(1)$ because dictionaries using hashing (like Python dictionaries and R environments).

```
function makedict(n)
  d=Dict{String,Int}()
  for i in 1:n
    push!(d, string(i) => i)
  end
  return d
end

## Make a large dictionary, with keys equal to strings representing integers.
d = makedict(n);
indstring = string(ind);
@btime d[indstring];
```

```
  42.118 ns (1 allocation: 16 bytes)
```

Finally, let's consider tuples. Lookup by index is quite slow, which is surprising as I was expecting it to be similar to lookup in the array, as I think the tuple in this case has values stored contiguously.

```
xt = Tuple(x);
@btime xt[ind];
```

```
49.232 ms (1 allocation: 16 bytes)
```

For named tuples, I'm not sure how realistic this is, since it would probably be a pain to create a large named tuple. But we see that lookup by name is slow, even though we are using a smaller tuple than the array and dictionary above.

```
## Set up a named tuple (this is very slow for large array, so use a subset).
dsub = makedict(100000);
xsub = x[1:100000];
names = Symbol.('x' .* keys(dsub));  # For this construction of tuple, the keys need to be symbols.
xtnamed = (;zip(names, xsub)...);
@btime xtnamed.x50000;
```

```
60.583  s (1 allocation: 16 bytes)
```

> **Developing a perspective on speed**
>
> Note that while all the individual operations above are fast in absolute terms for a single lookup, for simple operations, we generally want them to be really fast (e.g., order of nanoseconds) because we'll generally be doing a lot of such operations for any sizeable overall computation.

Performance tips

The Julia manual has an extensive section on performance.

We won't dive too deeply into all the complexity, but here are a few key tips, which mainly relate to writing in a way that is aware of the JIT compilation that will happen:

- Code for which performance is important should be inside a function, as this allows for JIT compilation.
- Avoid use of global variables that don't have a type, as that is hard to optimize since the type could change.
- The use of immutable objects can improve performance.
- Have functions always return the same type and avoid changing (or unknown) variable types within a function.