

Notes 5: Efficiency

Chris Paciorek

2025-02-04

Table of contents

Introduction	1
Timing	1
Profiling	2
Pre-allocation	3
Vectorization	5
Loop fusion	6
Memory allocation with loop fusion	7
Cases without loop fusion	8
Cache-aware programming and array storage	8
Store values contiguously in memory	10
Lookup speed	11
Performance tips	12

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.
```

0.082841 seconds (2.62 k allocations: 178.000 KiB, 11.25% compilation time)

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

0.079297 seconds

```
using BenchmarkTools
```

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

61.409 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 difficult. 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);
```

334.016 ms (2 allocations: 762.94 MiB)

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

1.691 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 $1+2+\dots+n = n(n+1)/2$ 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) / )
```

```
[-0.8053580347304067, 0.29822658211797026, 0.44145193792818144, 1.1918993408262208, -1.61158850801835]
```

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);
```

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

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

```
3.064 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 haven’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);
```

1.062 s (6 allocations: 2.24 GiB)

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

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

452.192 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
```



```
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);
```

39.277 ms (1 allocation: 896 bytes)

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

43.795 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.

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

135.965 ms (405 allocations: 610.36 MiB)

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

760.445 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.
```

32.860 s (3 allocations: 78.19 KiB)

100×100 Matrix{Float64}:

35.2411	27.4571	30.0888	24.9573	...	25.4787	26.3242	26.8453	27.4385
27.4571	35.2352	27.5602	24.6793		26.4031	23.9583	26.5808	27.3069
30.0888	27.5602	36.7827	25.0489		27.2198	25.5171	27.1006	28.3132
24.9573	24.6793	25.0489	30.5518		23.812	22.8682	24.0358	24.8542
25.1101	25.0275	25.2019	23.1299		23.1732	21.6282	23.9255	25.0931
24.9492	24.7618	26.6832	23.6232	...	23.9209	23.7042	24.9007	27.5235
27.6061	27.2086	28.3813	25.2452		25.6447	25.5787	27.0951	28.3547
27.4012	27.9134	28.992	24.6248		28.5521	25.4648	26.3295	29.0249
25.2387	26.2629	27.6689	22.2494		23.7467	22.368	25.631	26.2715
29.1681	28.9811	31.0503	27.1276		27.9643	28.6906	27.8429	29.9354
24.6922	23.3599	25.1643	22.4485	...	22.4468	22.9126	23.0442	25.0069
24.7518	23.8695	24.0303	20.1001		22.5691	21.7722	24.0761	24.8374
23.5739	24.3973	25.8065	22.2459		23.3258	22.3373	22.2234	25.3881
27.4036	27.2191	27.3575	24.8945		26.1581	24.8121	25.678	27.1647
25.1724	24.9983	25.8058	21.5679		23.824	24.2773	23.0613	25.1265
23.8596	23.9417	25.3964	20.8839	...	22.597	22.6916	23.4728	22.2023
25.2251	28.3095	27.7829	23.2355		24.2691	21.7259	25.3592	26.2605
29.8862	29.8705	30.6572	26.7172		28.9336	27.3684	28.4728	28.5583
25.0128	25.7503	24.2047	23.2712		23.2065	21.9484	22.8627	25.1435
24.9455	25.7858	25.9074	24.0675		24.3327	23.5105	24.5479	25.9077
25.6122	27.3248	27.4022	23.4455	...	26.9978	25.907	26.115	27.4218

25.4787	26.4031	27.2198	23.812	33.4213	26.5708	25.6988	26.9476
26.3242	23.9583	25.5171	22.8682	26.5708	32.0261	24.8115	25.2966
26.8453	26.5808	27.1006	24.0358	25.6988	24.8115	33.5558	27.8608
27.4385	27.3069	28.3132	24.8542	26.9476	25.2966	27.8608	38.2677

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

42.385 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];
```

20.084 ns (1 allocation: 16 bytes)

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

20.762 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];
```

39.558 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];
```

50.336 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;
```

60.586 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.