# 09. Julia is fast

October 19, 2018

## 1 Julia is fast

Very often, benchmarks are used to compare languages. These benchmarks can lead to long discussions, first as to exactly what is being benchmarked and secondly what explains the differences. These simple questions can sometimes get more complicated than you at first might imagine.

The purpose of this notebook is for you to see a simple benchmark for yourself. One can read the notebook and see what happened on the author's Macbook Pro with a 4-core Intel Core I7, or run the notebook yourself.

(This material began life as a wonderful lecture by Steven Johnson at MIT: https://github.com/stevengj/18S096/blob/master/lectures/lecture1/Boxes-and-registers.ipynb.)

## 2 Outline of this notebook

- Define the sum function
- Implementations & benchmarking of sum in...
  - C (hand-written)
  - C (hand-written with -ffast-math)
  - python (built-in)
  - python (numpy)
  - python (hand-written)
  - Julia (built-in)
  - Julia (hand-written)
  - Julia (hand-written with SIMD)
- Summary of benchmarks

## 3 `sum`: An easy enough function to understand

Consider the **sum** function `sum(a)`, which computes

$$\mathrm{sum}(a) = \sum_{i=1}^{n} a_i,$$

where $n$ is the length of `a`.

```
In [ ]: a = rand(10^7) # 1D vector of random numbers, uniform on [0,1)
In [ ]: sum(a)
```

The expected result is 0.5 * 10^7, since the mean of each entry is 0.5

## 4 Benchmarking a few ways in a few languages

```
In [ ]: @time sum(a)
```

```
In [ ]: @time sum(a)
```

```
In [ ]: @time sum(a)
```

The `@time` macro can yield noisy results, so it's not our best choice for benchmarking! Luckily, Julia has a `BenchmarkTools.jl` package to make benchmarking easy and accurate:

```
In [ ]: # using Pkg
        # Pkg.add("BenchmarkTools")
```

```
In [ ]: using BenchmarkTools
```

## 5 1. The C language

C is often considered the gold standard: difficult on the human, nice for the machine. Getting within a factor of 2 of C is often satisfying. Nonetheless, even within C, there are many kinds of optimizations possible that a naive C writer may or may not get the advantage of.

The current author does not speak C, so he does not read the cell below, but is happy to know that you can put C code in a Julia session, compile it, and run it. Note that the `"""` wrap a multi-line string.

```
In [ ]: using Libdl
        C_code = """
        #include <stddef.h>
        double c_sum(size_t n, double *X) {
            double s = 0.0;
            for (size_t i = 0; i < n; ++i) {
                s += X[i];
            }
            return s;
        }
        """

        const Clib = tempname()   # make a temporary file


        # compile to a shared library by piping C_code to gcc
        # (works only if you have gcc installed):

        open(`gcc -fPIC -O3 -msse3 -xc -shared -o $(Clib * "." * Libdl.dlext) -`, "w") do f
```

```
            print(f, C_code)
        end

        # define a Julia function that calls the C function:
        c_sum(X::Array{Float64}) = ccall(("c_sum", Clib), Float64, (Csize_t, Ptr{Float64}), leng
```

```
In [ ]: c_sum(a)
```

```
In [ ]: c_sum(a)  sum(a) # type \approx and then <TAB> to get the  symbolb
```

```
In [ ]: c_sum(a) - sum(a)
```

```
In [ ]:   # alias for the `isapprox` function
```

```
In [ ]: ?isapprox
```

We can now benchmark the C code directly from Julia:

```
In [ ]: c_bench = @benchmark c_sum($a)
```

```
In [ ]: println("C: Fastest time was $(minimum(c_bench.times) / 1e6) msec")
```

```
In [ ]: d = Dict()  # a "dictionary", i.e. an associative array
        d["C"] = minimum(c_bench.times) / 1e6  # in milliseconds
        d
```

```
In [ ]: using Plots
        gr()
```

```
In [ ]: using Statistics # bring in statistical support for standard deviations
        t = c_bench.times / 1e6 # times in milliseconds
        m,  = minimum(t), std(t)

        histogram(t, bins=500,
            xlim=(m - 0.01, m + ),
            xlabel="milliseconds", ylabel="count", label="")
```

## 6 2. C with -ffast-math

If we allow C to re-arrange the floating point operations, then it'll vectorize with SIMD (single instruction, multiple data) instructions.

```
In [ ]: const Clib_fastmath = tempname()   # make a temporary file

        # The same as above but with a -ffast-math flag added
        open(`gcc -fPIC -O3 -msse3 -xc -shared -ffast-math -o $(Clib_fastmath * "." * Libdl.dlex
            print(f, C_code)
        end

        # define a Julia function that calls the C function:
        c_sum_fastmath(X::Array{Float64}) = ccall(("c_sum", Clib_fastmath), Float64, (Csize_t, P
```

```
In [ ]: c_fastmath_bench = @benchmark $c_sum_fastmath($a)
```

```
In [ ]: d["C -ffast-math"] = minimum(c_fastmath_bench.times) / 1e6  # in milliseconds
```

## 7 3. Python's built in `sum`

The `PyCall` package provides a Julia interface to Python:

```
In [ ]: # using Pkg; Pkg.add("PyCall")
        using PyCall
```

```
In [ ]: # get the Python built-in "sum" function:
        pysum = pybuiltin("sum")
```

```
In [ ]: pysum(a)
```

```
In [ ]: pysum(a)  sum(a)
```

```
In [ ]: py_list_bench = @benchmark $pysum($a)
```

```
In [ ]: d["Python built-in"] = minimum(py_list_bench.times) / 1e6
        d
```

## 8 4. Python: `numpy`

### 8.1 Takes advantage of hardware "SIMD", but only works when it works.

`numpy` is an optimized C library, callable from Python. It may be installed within Julia as follows:

```
In [ ]: # using Pkg; Pkg.add("Conda")
        using Conda
```

```
In [ ]: # Conda.add("numpy")
```

```
In [ ]: numpy_sum = pyimport("numpy")["sum"]

        py_numpy_bench = @benchmark $numpy_sum($a)
```

```
In [ ]: numpy_sum(a)
```

```
In [ ]: numpy_sum(a)  sum(a)
```

```
In [ ]: d["Python numpy"] = minimum(py_numpy_bench.times) / 1e6
        d
```

## 9 5. Python, hand-written

```
In [ ]: py"""
        def py_sum(A):
            s = 0.0
            for a in A:
                s += a
            return s
        """

        sum_py = py"py_sum"
```

```
In [ ]: py_hand = @benchmark $sum_py($a)
```

```
In [ ]: sum_py(a)
```

```
In [ ]: sum_py(a)  sum(a)
```

```
In [ ]: d["Python hand-written"] = minimum(py_hand.times) / 1e6
        d
```

## 10 6. Julia (built-in)

### 10.1 Written directly in Julia, not in C!

```
In [ ]: @which sum(a)
```

```
In [ ]: j_bench = @benchmark sum($a)
```

```
In [ ]: d["Julia built-in"] = minimum(j_bench.times) / 1e6
        d
```

## 11 7. Julia (hand-written)

```
In [ ]: function mysum(A)
            s = 0.0 # s = zero(eltype(a))
            for a in A
                s += a
            end
            s
        end
```

```
In [ ]: j_bench_hand = @benchmark mysum($a)
```

```
In [ ]: d["Julia hand-written"] = minimum(j_bench_hand.times) / 1e6
        d
```

## 12 8. Julia (hand-written w. simd)

```
In [ ]: function mysum_simd(A)
            s = 0.0 # s = zero(eltype(A))
            @simd for a in A
                s += a
            end
            s
        end
```

```
In [ ]: j_bench_hand_simd = @benchmark mysum_simd($a)
```

```
In [ ]: mysum_simd(a)
```

```
In [ ]: d["Julia hand-written simd"] = minimum(j_bench_hand_simd.times) / 1e6
        d
```

## 13 Summary

```
In [ ]: for (key, value) in sort(collect(d), by=last)
            println(rpad(key, 25, "."), lpad(round(value; digits=1), 6, "."))
        end
```