Statistically significant performance testing of Julia scientific programming language

IOP Conf. Series: Journal of Physics: Conf. Series **1205** (2019) 012017 doi:10.1088/1742-6596/1205/1/012017

Statistically significant performance testing of Julia scientific programming language

M N Gevorkyan[1], A V Demidova[1], A V Korolkova[1] and D S Kulyabov[1,2]

[1]Department of Applied Probability and Informatics,
Peoples' Friendship University of Russia (RUDN University),
6 Miklukho-Maklaya str., Moscow, 117198, Russia
[2]Laboratory of Information Technologies, Joint Institute for Nuclear Research,
6 Joliot-Curie, Dubna, Moscow region, 141980, Russia

E-mail: `gevorkyan-mn@rudn.ru, demidova-av@rudn.ru, korolkova-av@rudn.ru, kulyabov-ds@rudn.ru`

Abstract. In this article, we compare the support for SIMD instructions for Julia and Fortran. The comparison is carried out according to the methodology described in work of T. Kalibera, R. E. Jones. The first part of the article gives a brief description of this technique. We emphasize on the practical implementation using Python, NumPy and SciPy. The second part of the article briefly discusses the syntactic capabilities of Fortran and Julia to work with SIMD processor extensions. Specific code snippets are given. Next, the performance of Julia and Fortran is compared for arithmetic operations on arrays of small length. The results are presented in tabular and graphical form.

1. Introduction

In this paper, we benchmark Fortran and Julia [1, 2, 3, 4, 5] SIMD instructions implementation. We carry out benchmarking guided by the methodology described in papers [6, 7]. This technique allows to obtain the statistically significant estimator of average time and to determine the optimal number of measurements.

In the first part of our paper, we provide the brief mathematical introduction in technique [6, 7]. We mainly focus on how to implement this technique using Python [8, 9] with Numpy [10], SciPy [11] and Matplotlib [12] libraries. Our source code is open and available at `https://bitbucket.org/mngev/fortran-vs-julia`.

In the second part of the article, we describe Fortran (GNU Fortran) and Julia languages SIMD vectorization capabilities and give some code snippets. In the last part of the paper we introduce benchmark results and give some summary.

2. Statistically significant estimator of program execution time

A naive approach to measuring the execution time of a program (or subroutine or function) is to repeatedly call the program and find the average time of its execution. Sometimes the first few runs of the program are not taken into account, as they are needed for warm-up. The number of execution is usually chosen heuristically. If we want to obtain statistically significant results, this approach cannot be considered satisfactory. In our work, we use the [6, 7] methodology.

These papers are freely available on the Internet, so we will focus on the implementation of the methods proposed in these works using the Python language and its libraries [13, 14]

2.1. Experiment's levels

Before starting an experiment, one should determine the maximum number of *experiment levels*. In our case, we will measure the execution time of the function. The first level is r_1 function calls inside the program, the second level is r_2 executions of program and the third is r_3 completions of the program source code. In general case it is possible to introduce any number of experiment levels.

At the first stage, a preliminary experiment with a heuristic choice of the number of levels $n+1$ and the number of tests r_i at each i-th level is carried out. Each test gives a value of the measurement time $X_{j_{n+1}j_n\dots j_2j_1}$, where $j_i = 1, \dots, r_i$. For example, if we measure the time spent on the third call of a function when the program is run for the second time using the fourth compiled executable file, it will be X_{423}. To store the results of measurements we used a multidimensional NumPy array. Every element of this array may be considered independent random number.

At each level, a one-dimensional sample of the obtained time measurements is considered.

Level $i = 1$. We fix the indexes of previous higher levels and get a sample of the following type:

$$\mathbf{X}_{j_{n+1}j_n\dots j_3j_2\bullet} = (X_{j_{n+1}j_n\dots j_2 1}, X_{j_{n+1}j_n\dots j_2 2}, \dots, X_{j_{n+1}j_n\dots j_2 r_1}).$$

The total number of such samples will be $r_{n+1} \cdot r_n \cdot \ldots \cdot r_3 \cdot r_2$ for all possible combinations of higher indexes. When using NumPy arrays to obtain, for example, $\mathbf{x}_{42\bullet}$, it is sufficient to take a slice by the last dimension: `x[3, 1,:]` (the indexing starts from zero).

Level $i = 2$. To study the results of the second level of execution, consider a sample of $\mathbf{x}_{j_{n+1}j_n\dots j_3j_2\bullet}$ and find its average value. As a result, for a fixed set of indexes $j_{n+1}j_n \dots j_3j_2$ we get a scalar value $\bar{X}_{j_{n+1}j_n\dots j_3j_2\bullet}$. By changing the index j_2 we get a sample for the second level of the experiment.

$$\bar{X}_{j_{n+1}j_n\dots j_3j_2\bullet} = \mathrm{M}[\mathbf{X}_{j_{n+1}j_n\dots j_3j_2\bullet}] = \frac{1}{r_1}\sum_{j_1=1}^{r_1} X_{j_{n+1}j_n\dots j_2j_1},$$

$$\mathbf{X}_{j_{n+1}j_n\dots j_3\bullet\bullet} = (\bar{X}_{j_{n+1}j_n\dots j_3 1\bullet}, \bar{X}_{j_{n+1}j_n\dots j_3 2\bullet}, \dots, \bar{X}_{j_{n+1}j_n\dots j_3 r_2\bullet})$$

The total number of such samples will be $r_{n+1}\cdot r_n\cdot\ldots\cdot r_3$ for all possible combinations of indexes greater than 2. When you are working with NumPy arrays, the calculation of $\bar{X}_{j_{n+1}j_n\dots j_3j_2\bullet}$ is made with functions `mean` by applying it to the last dimension: `np.mean (X, axis=-1)`. The result is an array with the number of dimensions reduced by one. For the following levels, similar actions have to be taken.

Level $i = n$.

$$\bar{X}_{j_{n+1}j_n\underbrace{\bullet\dots\bullet}_{n-1}} = \mathrm{M}[\mathbf{X}_{j_{n+1}j_n\underbrace{\bullet\dots\bullet}_{n-1}}] = \frac{1}{r_{n-1}}\sum_{j_{n-1}=1}^{r_{n-1}} \bar{X}_{j_{n+1}j_nj_{n-1}\underbrace{\bullet\dots\bullet}_{n-2}}$$

$$\mathbf{X}_{j_{n+1}\underbrace{\bullet\dots\bullet}_{n}} = (\bar{X}_{j_{n+1}1\underbrace{\bullet\dots\bullet}_{n-1}}, \bar{X}_{j_{n+1}2\underbrace{\bullet\dots\bullet}_{n-1}}, \dots, \bar{X}_{j_{n+1}r_n\underbrace{\bullet\dots\bullet}_{n-1}})$$

Level $n+1$

$$\bar{X}_{j_{n+1}\underbrace{\bullet\dots\bullet}_{n}} = \mathrm{M}[\mathbf{X}_{j_{n+1}\underbrace{\bullet\dots\bullet}_{n}}] = \frac{1}{r_n}\sum_{j_n=1}^{r_n} X_{j_{n+1}j_n\underbrace{\bullet\dots\bullet}_{n-1}}$$

$$\mathbf{X}_{\underbrace{\bullet\dots\bullet}_{n+1}} = (\bar{X}_1\underbrace{\bullet\dots\bullet}_{n}, \bar{X}_2\underbrace{\bullet\dots\bullet}_{n}, \dots, \bar{X}_{r_{n+1}}\underbrace{\bullet\dots\bullet}_{n})$$

Finally, to get the mean time for all tests, we should find the mean for the entire data set (the grand mean).

$$\bar{X} = \mathrm{M}[\bar{X}_{\underbrace{\bullet\ldots\bullet}_{n+1}}] = \frac{1}{r_{n+1}} \sum_{j_{n+1}=1}^{r_{n+1}} X_{j_{n+1}} \underbrace{\bullet\ldots\bullet}_{n}.$$

2.2. Visual analysis

After the measurements are completed, one should proceed to a visual evaluation of the data, which will give an opportunity to estimate the number of pre-runs required for the system warm-up. We denote the number of pre-runs for each level by $c_1, c_2, \ldots, c_{r_i}$. For each level i one should draw three plots.

- The first plot is the run-sequence plot. The horizontal axis represents the test number from 1 to r_i, and the vertical axis — $X_1, \ldots, X_{r_i}$. Additionally, one may build a similar plot with mixed values $X_1, \ldots, X_{r_i}$. The analysis of the resulting image will help to determine how many pre-runs the system requires to warm-up.
- The second plot is the lag-plot. One axis represents the values $X_1, \ldots, X_{r_i-l}$ and on the second axis — $X_l, \ldots, X_{r_i}$, where the integer l specifies the lag. If the numbers $X_1, \ldots, X_{r_i}$ are independent random values, the points on the chart will be placed chaotically, without visible structure.
- The third plot is the auto-correlation function plot (ACF-plot). It allows to estimate the correlation between a sample of $\mathbf{X} = X_1, \ldots, X_{r_i-l}$ and its version with lag $\mathbf{X}_l = X_l, \ldots, X_{r_i}$. The value of the autocorrelation is determined by the formula

$$\mathrm{cor}(\mathbf{X}, \mathbf{X}_l) = \frac{\sum_{k=1}^{r_i-l} (X_k - \bar{X})(X_{l+k-1} - \bar{X}_l)}{\sqrt{\sum_{k=1}^{r_i-l} (X_k - \bar{Y})^2 \sum_{k=l}^{r_i} (X_k - \bar{X}_l)^2}}$$

$$\bar{X} = \frac{1}{r_i - l} \sum_{k=1}^{r_i-l} X_k, \quad \bar{X}_l = \frac{1}{r_i - l} \sum_{k=l}^{r_i} X_k.$$

  To draw the plot along the horizontal axis, the lag values l are plotted, and along the vertical axis, the values $\mathrm{cor}(\mathbf{X}, \mathbf{X}_l)$ for each corresponding l are plotted.

The last two plots are called correlograms and allow us to determine whether the available time measurements can be considered independently distributed random variables.

To build the first plot, you may use the `scatter` function of the Matplotlib library to visualize points cloud. To mix the elements in the array, the `np.random.shuffle` function is used.

By using the same `scatter` function in conjunction with array slices we can build the second plot. The values `X[l:]` are plotted on one axis and the values `X[:-l]` are plotted on the other axis, where `l` specifies the lag. It is also useful to draw the bisector of the first quarter of the coordinate axes.

To plot the autocorrelation graph, the use of the `acorr` function will give an incorrect result, since it uses a different algorithm for calculating autocorrelation in signal processing. So the calculation of autocorrelation must be implemented independently. The result of its work can be displayed on the chart using the `vlines` function.

2.3. Biased and unbiased estimators

S_i^2 is biased estimator for variance at level i. It is calculated by following formulas

$$S_1^2 = \frac{1}{\prod\limits_{k=2}^{n+1} r_k} \frac{1}{r_1 - 1} \sum_{j_{n+1}=1}^{r_{n+1}} \cdots \sum_{j_1=1}^{r_1} (X_{j_{n+1} j_n \ldots j_2 j_1} - \bar{X}_{j_{n-1} j_n \ldots j_2 \bullet})^2;$$

$$S_i^2 = \frac{1}{\prod\limits_{k=i+1}^{n+1} r_k} \frac{1}{r_i - 1} \sum_{j_{n+1}=1}^{r_{n+1}} \cdots \sum_{j_i=1}^{r_i} (\bar{X}_{j_{n+1} j_n \ldots j_i \underbrace{\bullet \ldots \bullet}_{i-1}} - \bar{X}_{j_{n+1} j_n \ldots j_{i+1} \underbrace{\bullet \ldots \bullet}_{i}})^2, \ i = 2, \ldots, n;$$

$$S_{n+1}^2 = \frac{1}{r_{n+1} - 1} \sum_{j_{n+1}=1}^{r_{n+1}} (X_{j_{n+1} \underbrace{\bullet \ldots \bullet}_{n}} - \bar{X}_{\underbrace{\bullet \ldots \bullet}_{n+1}})^2;$$

Despite the cumbersome mathematical formulas, the computation of s_i^2 in terms of NumPy arrays is a trivial task. For example, let us measure time with three levels of experiment. As a result of measurements we obtain a three-dimensional array `X`. To calculate S_i^2, one need to use `mean` function for 2 and 3 dimensions, and then `var` function to find the unbiased sample variance

```
res = np.mean(X, axis=(-2, -1))
res = np.var(res, ddof=1, axis=-1)
```

where the parameter `axis=(-2, -1)` indicates that the function should use only last tow dimensions of the array, and the parameter `ddof=1` indicates that we need an unbiased sample variance.

When we have S_i^2 we can use iterative process to calculate the unbiased estimators T_i^2:

$$T_1^2 = S_1^2, \ \ T_i^2 = S_i^2 - \frac{S_{i+1}^2}{r_{i-1}}, \ i = 2, \ldots, n+1.$$

Estimators T_i^2 allow to calculate the optimal number of repetitions for each level of the experiment

$$n_1 = \sqrt{c_1 \frac{T_1^2}{T_2^2}}, \ n_i = \sqrt{\frac{c_i}{c_{i-1}} \frac{T_i^2}{T_{i+1}^2}},$$

where c_i is the number of repetitions, needed for the level i warm-up.

3. SIMD vector operations in Fortran and Julia

3.1. SIMD extensions

SIMD extensions (single instruction stream / multiple data stream) can be found in most modern processors. SIMD extensions are a set of special registers and instructions that can be applied to these registers in parallel mode. The most common modern SIMD extensions are AVX (Advanced Vector Extensions) of different revisions. For the measurements we used Intel Haswell processor with AVX2 support, which gives 16 registers with a total volume of 256 bits. Modern AMD Zen processors also support AVX2 instructions.

SIMD vectorization is often the easiest way to improve the performance of numerical methods for solving systems of differential equations [15]

3.2. Syntax for SIMD operations in Fortran and Julia

Let's consider several ways how to add one array to another in Fortran and Julia languages. Let's begin from Fortran and set three dynamic arrays and then fill them with random numbers.

```
integer, parameter :: array_size = 8
real, allocatable, dimension(:) :: V, U, Res
integer :: i
allocate(A(array_size), B(array_size), C(array_size))
call random_number(V); call random_number(U)
```

In the first example, let's add two arrays elementwise in a loop and write the result to the third array

```
do i = 1,array_size,1
  Res(i) = V(i) + U(i)
end do
```

One may also use the `concurrent` statement added to the Fortran 2008 [16, 17, 18] standard to explicitly point out the loop iterations independence:

```
do concurrent(i = 1:array_size,1)
  Res(i) = V(i) + U(i)
end do
```

In the second example, we perform the same summation, but without cycles, using so-called vector notation

```
Res = V + U
```

We intentionally use dynamic arrays in order to provide a more fair comparison of languages, as in the Julia standard arrays can change size during execution.

There are three options for similar actions in Julia. Two of them are comparable with Fortran:

```
const array_size = 8
V = Vector{Float32}(array_size)
U = Vector{Float32}(array_size)
Res = Vector{Float32}(array_size)
rand(V); rand(U); rand(Res)
for i in 1:array_size
  Res[i] = V[i] + U[i]  # iterative summation
end
Res = V + U  # same without loop
```

The third option allows using `@simd` and `@inbounds` macros to explicitly specify the usage of SIMD instructions.

```
@simd for i in 1:20
  @inbounds Res[i] = V[i] + U[i]
end
```

3.3. Benchmarking

Benchmarking is carried out for GNU Fortran [19] version 7.3.0. To get a log file about the vectorization process, we use the `-fopt-info-vec-all=file.log` command line option during the compilation. We implement above examples as separate functions and use the `cpu_time` intrinsic subroutine to measure time.

Tests are performed for arrays with 32-bits floating point numbers. The length of the array is calculated from the total volume of AVX registers (256 bits). The division by 32 bits gives

exact 8. The time required for 10^4 function calls is measured. We perform 3 experiment levels, where $r_1 = 100$, $r_2 = 20$ and $r_3 = 20$. The total amount is $100 \cdot 20 \cdot 20$ measurements.

We also perform separate measurements for each of the three possible optimization levels: `O1`, `O2` and `-O3`.

	Overall mean		**Confident interval sizes**		**Optimum**	
	Iter.	Vect.	Iter.	Vect.	Iter.	Vect.
`O1`	0.000436	0.000246	0.000009	0.000011	34	27
`O2`	0.000405	0.000374	0.000010	0.000009	41	29
`O3`	0.000400	0.000379	0.000008	0.000009	33	36

Table 1. Results for Fortran. Confident interval was calculated for $\alpha = 0.01$

The benchmark results are organized in three tables 1. The first column of each table contains measurements for iterative summation and the second column contains data for vectorized summation. The last table contains the optimal number of repetitions for each experiment level which are calculated as described in first part of this paper. For overall mean we also plot bar charts 1.

The obtained measurement data show that when using the first level of optimization, the compiler automatically applies SIMD instructions to those functions where non-index summation was used. At a higher level of optimization, the compiler automatically detects loop independence and uses SIMD instructions even for iterative summation.

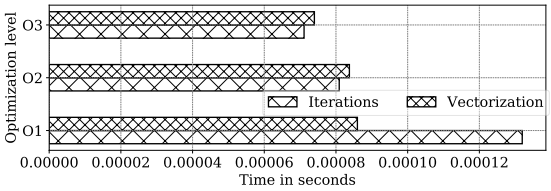

Figure 1. Overall mean from Table 1

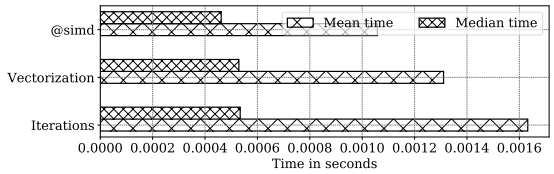

Figure 2. Overallmean from Table 2

Julia language is implemented as JIT-compiler, so there is no explicit compilation stage. So we left only 20 program executions and 100 functions calls. Also, there are no levels of optimization. The results of measurements are presented in the table 2, and the plot 2.

	Iterations	**Vectorization**	`@simd`
Overall mean	0.00163201	0.00131059	0.00105772
Conf. interval	0.0011087924109	0.00087504387	0.00071035994
Optimal	11	11	11

Table 2. Results for Julia. Confident interval was calculated for $\alpha = 0.01$

The plot shows that the average time of Julia program is almost two orders of magnitude grater than Fortran. However, the median time is less than the average Fortran program execution time by an order of magnitude. If we consider the run-sequence plot, it becomes clear that a some values of measurements differ greatly from the average time. It is these anomalous measurements that lead to an increase in the average time.

As mentioned in the official documentation of the Julia language, the use of non-index operations in the Julia does not provide any significant increase in productivity (see median time). However, the use of the `@simd` macro allows to get slightly better loop performance.

4. Conclusion

In this article, we have compared the implementation of SIMD extensions in the GNU Fortran standard compiler and language Julia JIT compiler. Julia language in showed worse results, but there are a few caveats to be made.

First of all, the Julia language is in the process of intensive development and has not yet reached even the first version. Therefore, the results of measurements in any case can not be considered as final.

Also Julia language implements a lot of high level features from interpreting languages. Due to this, the development of software on Julia is potentially faster.

Acknowledgment

The publication has been prepared with the support of the "RUDN University Program 5-100" and funded by Russian Foundation for Basic Research (RFBR) according to the research project No 16-07-00556.

[1] Bezanson J, Edelman A, Karpinski S and Shah V B 2017 *SIAM Review* **59** 65–98
[2] Bezanson J, Karpinski S, Shah V B and Edelman A 2012 (*Preprint* `1209.5145`)
[3] Kulyabov D, Gevorkyan M, Korolkova A and Sevastianov L 2017 *CEUR Workshop Proceedings* **1995** 93–99
[4] Balbaert I, Sengupta A and Sherrington M 2016 *Julia: High Performance Programming* ISBN 9781787125704
[5] Kwon C 2016 *Julia Programming for Operations Research: A Primer on Computing* ISBN 9781533328793
[6] Kalibera T and Jones R E 2012 Quantifying Performance Changes with Effect Size Confidence Intervals Technical Report 4–12 University of Kent URL `http://www.cs.kent.ac.uk/pubs/2012/3233`
[7] 2013 *Rigorous Benchmarking in Reasonable Time* (New York: ACM)
[8] Rossum G 1995 Python reference manual Tech. rep. Amsterdam, The Netherlands, The Netherlands
[9] 2018 Python home site URL `https://www.python.org/`
[10] 2018 Numpy home site URL `http://www.numpy.org//`
[11] Jones E, Oliphant T, Peterson P *et al.* 2001– SciPy: Open source scientific tools for Python URL `http://www.scipy.org/`
[12] 2018 Matplotlib home site URL `https://matplotlib.org/`
[13] Unpingco J 2016 *Python for Probability, Statistics, and Machine Learning* (Springer International Publishing) ISBN 9783319307176
[14] Haslwanter T 2016 *An Introduction to Statistics with Python: With Applications in the Life Sciences* Statistics and Computing (Springer International Publishing) ISBN 9783319283166
[15] Gevorkyan M N, Demidova A V, Korolkova A V and Kulyabov D S 2018 *Distributed Computer and Communication Networks* (*Communications in Computer and Information Science* vol 919) ed Vishnevskiy V M and Kozyrev D V (Cham: Springer International Publishing) pp 532–546 ISBN 978-3-319-99447-5
[16] Brainerd W S 2015 *Guide to Fortran 2008 Programming* (London Heidelberg New York Dordrecht: Springer) ISBN 9781447167594
[17] Hanson R J and Hopkins T 2013 *Numerical computing with modern Fortran* (Philadelphia: SIAM) ISBN 9781611973112
[18] Chapman S J 2018 *Fortran for Scientists and Engineers* (New York: McGraw-Hill Education) ISBN 9780073385891
[19] 2018 Gnu fortran URL `https://gcc.gnu.org/fortran/`