Performance comparison of Julia and Python programming languages on GPU and CPU

M. MAJ
mateuszmaj2@mon.gov.pl

Command of the Cyberspace Defense Forces Component
Gen. T. Buka Str. 1, 05-119 Legionowo, Poland

The paper presents research on the time performance of each language when performing operations on the CPU (central processing unit) and GPU (graphics processing unit). This article should be considered as a coherent continuation of the first part [18]. The paper has been divided due to existing editorial criteria. The work includes descriptions of the technologies used and the algorithms that served the research purposes. Algorithms prepared with care to maintain equal use of functions by both programming languages based on block diagrams and pseudocode are presented. The paper first presents the research carried out on the basis of sorting algorithms implemented by the Author and executed on the processor. The next part of the paper presents research verifying the performance of Julia and Python programming languages on graphics cards, with a special focus on the implementation of matrix operations. The article is concluded with a discussion of the time performance of each programming language.

Keywords: Python, Julia, comparison, performance.

DOI: 10.5604/01.3001.0055.5704

1. Introduction

With the development of technology, there is an increasing demand for testing and checking the performance of new tools (with respect to widely used spread programming languages).

Julia, whose publication of the first stable version (Julia 1.0) dates back to 2018 [17] (please don't be confused with the first presentation of the Julia 0.1 language in 2012) [3], is a relatively young programming language compared to Python, whose stable versions have been released approximately every six to eighteen months or so since 1991 [1] (even taking into account the researched Python 3, which was presented to the world in 2008).

The article will test the each language's capabilities by implementing algorithms in a computer environment that is described in more detail later in the paper.

All abbreviations and simplified terms employed in this manuscript are fully expanded and defined in the "List of Commonly Used Abbreviations and Terms" provided at the end of the document.

This article is a continuation of the comparative analysis [18] and the author of the articles recognizes that they should be treated as a coherent whole. The content of this work complements the brads and observations of the article which, for reasons of meeting publishing criteria, has been divided into two parts.

1.1. Description of the used tools

The computer environment used for the following tests includes an intermediate development kit in the form of Microsoft Windows 10 Home 22H2 x64-based PC architecture. Intel(R) Core(TM) i5-9300H CPU @ 2.40GHz, 2400 M Hz, Cores: 4, Logic processors: 8. Installed physical memory (RAM) 16.0 GB. Motherboard manufacturer and product HP 8640. Graphics card NVIDIA GeForce GTX 1050, and most importantly the versions of the languages used, namely Julia version 1.8.5 and Python 3.11.2. All programs was called in Visual Studio Code 1.79.0 0 [5], [6], [7] environment without INTERNET connection.

The research conducted on CPU was based on sorting algorithms further referred to as bubble sort, quick sort and insertions sort. The simplicity of the programs is important and has a significant impact on veracity of result conducted tests. The implementation of sorting algorithms in Python and Julia programming languages should correspond to each other (as recommended by White [4]). This means that they must use the same functions and methods for reliable analysis and obtaining reliable research results. For this purpose, the algorithms were prepared based on previously prepared block diagrams, as well as

the necessary pseudocodes. The sorting algorithms check the efficiency of the execution times of the algorithms on the CPU based on previously prepared input data that will be executed ten times on the computing unit and given in the article as calculated values according to the arithmetic mean.

Matrix operations were used to check the performance of each language using the capabilities of graphics cards based on several necessary libraries. The research was conducted based on four libraries. Three of them are written in Python and those are TensorFlow, CuPy and PyTorch and one written in Julia named CUDA (Compute Unified Device Architecture). All algorithms used in this work were carefully written while maintaining corresponding structures and methodology for both languages. Own research conducted on GPU was performed based on matrix multiplication operations as well as data conversion to arrays understandable by GPU. They differ for each library, so they were included in the measurement time. [16], [14], [13], [15].

The first library on which measurements were performed is CuPy. It is an open-source library used to work with Graphics Processing Unit in Python using CUDA technology. It is often used as a replacement for the NumPy library as its alternative using the capabilities of the graphics processor for numerical computing. Due to the similarities of the API interface to the NumPy library, transferring code to GPU is quite simple. CuPy allows to define kernels (own CUDA kernels for specific operations). CuPy support using a wide range of mathematical calculations such as operations on matrices for example multiplication, transposition or handling sparse matrices CSR, CSC, COO used in large systems of linear equations and other numerical calculations. Using cuFFT allows to perform Fourier operations and cuRAND makes it possible to generate pseudo-random numbers. CuPy is also widely used in scientific work as well as machine learning (as a backend for the Chainer library). CuPy is based on several key components, for example there are CuPy arrays named as “cupy.ndarrray” which are equivalents of the library NumPy arrays names “numpy.ndarray” that are stored in GPU memory. CuPy works with many Nvidia libraries, including cuDNN. However, it is not essential for operations on GPU. Installing CuPy is quite simple and can be done using pip. This library does not support running programs on CPU [16].

TensorFlow is another Python library developed by Google. It is open-source and designed for training and deploying machine learning models. The best way of using it is training deep neural networks. The library supports both CPU and GPU as well as TPU (Tensor Processing Units). When computing on GPU it is necessary to use externals libraries as cuDNN. It has a high performance API for production model deployment. The library can be used on various platforms thanks to TensorFlow Lite (mobile devices) and TensorFlow Serving (servers). It is used in building deep learning models such as neural networks (NN), recurrent neural networks (RNN) or convolutional neural networks (CNN). Since TensorFlow is a complex tool that makes it much difficult for beginners to use [14].

The last library used for this research, written in Python is PyTorch, which was designed to dynamically build and train deep learning models mainly developed by Facebook’s AI. It is particularly appreciated for its intuitiveness and simplicity. It is widely used in projects related to image processing (CV) or natural language (NLP). It is possible to create dynamic computational graphs (defined on the fly), which makes debugging models much easier. The API is quite simple and PyTorch supports computing units in the form of CPU and GPU. It is much simpler than TensorFlow but offers a smaller scope of use [13].

Tool used to create and run GPU computing programs in Julia is the CUDA.jl library. It was designed to create GPU applications based on the CUDA platform. It enables interaction with the GPU, using its computing power in scientific, engineering and machine learning applications. It is widely used in improving the performance of scientific computing, image processing or numerical modelling. CUDA.jl allows building machine learning models, provides support for low-level programming in CUDA and for higher--level matrix operations. The library automatically manages GPU memory, which improves the readability of the code and simplifies its use. It allows to write your own CUDA kernels using libraries such as BLAS or LAPACK on the GPU. Also allows to perform linear operations, Fourier transformations and much more. It achieves particularly satisfactory results for its own CUDA kernels. Full control over the GPU is provided to the user by the CUDA.jl library, which allows the code to be fine-tuned for performance, making it a particularly attractive tool for more complex projects [2], [15].

2. Results of own research

2.1. Algorithms

The author's algorithms used during the research on the bubble sort algorithm are presented as Code 1 and Code 2.

Code 1. Bubble sort algorithm in the Python language

```
    def bubbleSort(list):
        t1 = datetime.now()
        leng = len(list)
        for i in range(leng-1):
            for j in range(1,leng):
                if list[j-1]>list[j]:
                    tmp = list[j-1]
                    list[j-1] = list[j]
                    list[j] = tmp
        t2 = datetime.now()
        with open("executionTime.txt", "a+") as f:
            f.write(f"{t2 - t1}\n")
```

Code 2. Bubble sort algorithm in the Julia language

```
function bubbleSort(list)
    t1 = time()
    len = length(list)
    for i = 1:len-1
        for j = 2:len
            if list[j-1] > list[j]
            tmp = list[j-1]
            list[j-1] = list[j]
            list[j] = tmp
            end
        end
    end
    t2 = time()
    open("executionTime.txt","a+")do f
        write(f, "$(t2 - t1)\n")
    end
end
```

Code 3. Insertion sort algorithm in the Julia language

```
    function insertionSort(list)
        t1 = time()
        for i in 2:length(list)
            temp = list[i]
            j = i - 1
            while j >= 1 && list[j] > temp
                list[j + 1] = list[j]
                j -= 1
            end
            list[j + 1] = temp
        end
        t2 = time()
        open("executionTime.txt", "a+") do f
            write(f, "$(t2 - t1)\n")
        end
    end
```

Code 4. Insertion sort algorithm in the Python language

```
from datetime import datetime
def instertionSort(array):
    t1 = datetime.now()
    for i in range(1,len(array)):
        temp=array[i]
        j=i-1
        while j>=0 and array[j]>temp:
            array[j+1]=array[j]
            j-=1
        array[j+1]=temp
    t2 = datetime.now()
    with open("executionTime.txt", "a+") as f:
        f.write(f"{t2 - t1}\n")
```

Code 5. Quicksort algorithm in the Julia language

```
function partition(array, low, high)
    pivot = array[high]
    i = low -1
    for j in low:high-1
        if array[j] <= pivot
            i = i + 1
            array[i], array[j] = array[j], array[i]
        end
    end
    array[i + 1], array[high] = array[high], array[i + 1]
    return i + 1
end
function quickSort(array, low, high)
    if low < high
        pi = partition(array, low, high)
        quickSort(array, low, pi - 1)
        quickSort(array, pi + 1, high)
    end
end
```

Code 6. Quicksort algorithm in the Python language

```
def partition(array, low, high):
  pivot = array[high]
  i = low - 1
  for j in range(low, high):
    if array[j] <= pivot:
      i = i + 1
      (array[i], array[j]) = (array[j], array[i])
  (array[i + 1], array[high]) = (array[high], array[i + 1])
  return i + 1
def quickSort(array, low, high):
  if low < high:
    pi = partition(array, low, high)
    quickSort(array, low, pi - 1)
    quickSort(array, pi + 1, high)
```

Code 7. Matrix multiplication on the GPU using CUDA in the Julia language

```
function gpu_matrix_cuda(matrix1, matrix2)
    j_matrix1 = CuArray(matrix1)
    j_matrix2 = CuArray(matrix2)
    j_result = j_matrix1 * j_matrix2
   return Matrix(j_result)
end
```

Code 8. Matrix multiplication on the GPU using CuPy in the Python language

```
def gpu_matrix_multiply_cupy(matrix1, matrix2):
    c_matrix1 = cp.array(matrix1)
    c_matrix2 = cp.array(matrix2)
    c_result = cp.dot(c_matrix1, c_matrix2)
    return cp.asnumpy(d_result)
```

Code 9. Matrix multiplication on the GPU using PyTorch in the Python language

```
def gpu_matrix_multiply_torch(matrix1, matrix2):
    t_matrix1 = torch.tensor(matrix1, device='cuda')
    t_matrix2 = torch.tensor(matrix2, device='cuda')
    t_result = torch.matmul(t_matrix1, t_matrix2)
    return t_result.cpu().numpy()
```

Code 10. Matrix multiplication on the GPU using TensorFlow in the Python language

```
def gpu_matrix_multiply_tensorflow(matrix1, matrix2):
    with tf.device('/GPU:0'):
        tf_matrix1 = tf.convert_to_tensor(matrix1)
        tf_matrix2 = tf.convert_to_tensor(matrix2)
        tf_result = tf.linalg.matmul(tf_matrix1, tf_matrix2)
    return tf_result.numpy()
```

The bubble sort algorithm is characterized by a constant number of performed operations (regardless of the diversity of the input data set, the algorithm will perform a constant number of comparisons), the constant computational complexity of the algorithm (in the O notation it is $O(n^2)$) [12]. The name of the bubble sort algorithm is closely related to the analogy caused by the idea that larger elements are thrown (float) to the top of the list, similarly to bubbles in a soda breathing to the surface of a liquid. The algorithm will check whether any of the elements remains unsorted a given number of times (equal to the size of the input data array). This procedure will have a significant impact on extending the sorting time and will ensure the same number of operations performed by each of the sorting algorithms [10], [12].

The insertion sort algorithm is one of the most common sorting algorithms. The algorithm achieves its highest efficiency when working on pre-sorted sets. The insertion sort algorithm is stable and its complexity depends on the number of inversions in the permutation. The time complexity of the algorithm for the O notation is: $O(n^2)$ [12]. This makes it a suitable algorithm for the research [10], [11], [12]. The author's algorithms used during the research on the sort--by-insertion algorithm are presented as Code 3 and Code 4.

The quicksort algorithm is one of the most popular and fastest sorting algorithms of all time. This solution was developed by Tony Hoare in 1959 and first published in 1961, over 60 years before this article was written. The operation of the algorithm is simple, which translates into widespread distribution and use in the implementation of standard libraries of many programming languages. The used quicksort algorithm includes a call to recursion. All these elements make the quicksort algorithm an attractive candidate for research. This sorting algorithm is based on the "divide and conquer" approach, which can be divided into three main elements: DIVIDE – the main problem is divided into subproblems CONQUER – finding a solution to the subproblems JOIN – the solutions of the subproblems are combined into the solution to the main problem [12]. The programs prepared for the study of the quicksort algorithm in Julia and Python languages were presented as Code 5 and Code 6.

The matrix operation algorithm involves multiplying two matrices to obtain the result matrix. The operations are performed on two arrays A and B understood by the GPU, where the data is arranged linearly in memory or in columns depending on the system and programming language. The computer immediately allocates memory space for the result matrix C. The GPU then divides the input data into smaller blocks and assigns them to threads. Each thread is responsible for calculating one element of the result matrix c_{ij}. Each element of the result matrix requires iteration through row "i" of matrix A and iteration through column "j" of matrix B. Finally, the element c_{ij} is obtained from the sum of the products $c_{ij} = a_{ik} \cdot b_{kj}$. All these

operations are distributed in parallel between GPU threads. The computational complexity of matrix multiplication is $O(m \cdot n \cdot p)$. Where m m, n, p are the dimensions of the matrices $A(m \text{ x } n)$ and $B(n \text{ x } p)$. Matrix multiplication allows us to test languages in the context of working on GPUs due to its complexity and the possibility of parallelizing the algorithm.

The programs prepared to study the matrix multiplication algorithm on the GPU executed for CUDA, CuPy, PyTorch and TensorFlow libraries are presented as Code 7, Code 8, Code 9 and Code 10.

2.2. Summary of results

Each of the algorithms obtained different results, which is why they will be discussed separately based on tabular and graphical summaries. First, we will discuss the sorting algorithms executed on the CPU.

Table 1 presents the results of research on the insertion sort algorithm. Additionally, in order to better visualize the data, the ratio of the execution times of the insertion sorting algorithm in Julia and Python was presented. This allows it easier to see the difference in the execution time of each of the algorithms. The column labelled "Data size" lists the input sizes – that is, the number of elements to be sorted. A *k* suffix is added merely for readability all figures. For the same reason, concise symbols (e.g. t_{pi} stands for time for insertion sort algorithm tested in Python) were used in other columns they correspond to Equations (1) and (2). Other tables were marked in the same way.

The results presented at Table 1 can be expressed by models expressed by the formulas appropriately for the Julia (1) and Python (2) languages. Both models are power functions that have achieved data forecasting efficiency of over 95%

$$t_{ji} = (4{,}25 \cdot 10^{-11}) \cdot k^2 \quad (1)$$

$$t_{pi} = (1{,}97 \cdot 10^{-8}) \cdot k^2 \quad (2)$$

The numerical parameters presented in Table 1 can be clearly outlined by the quadratic function for each programming language. This indicates that performance increases almost in the same way for Julia as for Python, as well as the stability of the algorithm. A similar pattern can be seen in the remaining columns: their values are stable and clearly show that result Julia outperforms Python. Table 1 shows that the Julia language coped almost five times better than the Python language. On average, the program with the .jl extension needed 21% of the time needed to complete the same task in relation to the algorithm written in Python. The results are very stable for each data set. The amount of time needed to sort subsequent amounts of input data grows proportionally to the number of iterations for both programs. Moreover, the difference in the percentage share of time results between files prepared in the Julia language and those prepared in the Python language is one percentage point (21 ± 1[%]).

Figure 1 presents a comparison of the results of insertion sort for programs prepared in each described programming language. As it was said all figures contains suffix (i, e, k, t, tpi, tji). According to the presented results (Table 1), their graphic visualization in the form of a chart Figure 1 presents a constant (percentage) difference in the size of time results needed to perform the same tasks. For large numbers of executed iterations, the discrepancy in the time results of the algorithms' execution is close to 25 seconds in favor of the algorithm written in Julia. In percentage terms a constant difference can be observed between the execution of algorithms with different extensions.

The results of the research on the bubble sort algorithm can be observed in Table 2. The research was conducted on the same input data as the insertion sort algorithm operated on. Column named "Julia/Python" contains percentage score of the results obtained by the algorithm written in Julia compared to the algorithm written in Python.

Extrapolating the column named "Julia/ Python" more complex algorithms or larger data sets and assuming identical algorithmic complexity a computation that would occupy Python for 100 seconds is expected to finish in 24 seconds when implemented in Julia.

The results presented in Table 2 show that the implementation of the bubble sort task in Julia was more than four times more efficient than the solution implemented in Python (it can be seen in last column). The sorting results in the percentage representation are constant regardless of the amount of data. All results in Julia constitute an average of 24% of the time executing the sort in Python. Only one result deviated from the average value and that by only one percentage point. 0 presents a graphical comparison of the results of bubble sort algorithm.

The shape of the graph (Figure 2) is very similar to that presented in Figure 1. Definitely better execution times of the algorithms can be stated in Favor of Julia.

Tab. 1. Summary of the results for insertion sort algorithm

Data size (k)	Execution time [s]		Julia / Python [%]	Python/ Julia
	Python t_{pi}	Julia t_{ji}		
5000	0,4626	0,1008	22	4,6
7000	0,9304	0,1944	21	4,8
9000	1,5435	0,3175	21	4,9
11000	2,3075	0,4826	21	4,8
13000	3,2607	0,6932	21	4,7
15000	4,3472	0,9447	22	4,6
20000	7,7386	1,663	21	4,7
30000	17,6223	3,8494	22	4,6
40000	31,5808	6,7939	22	4,6

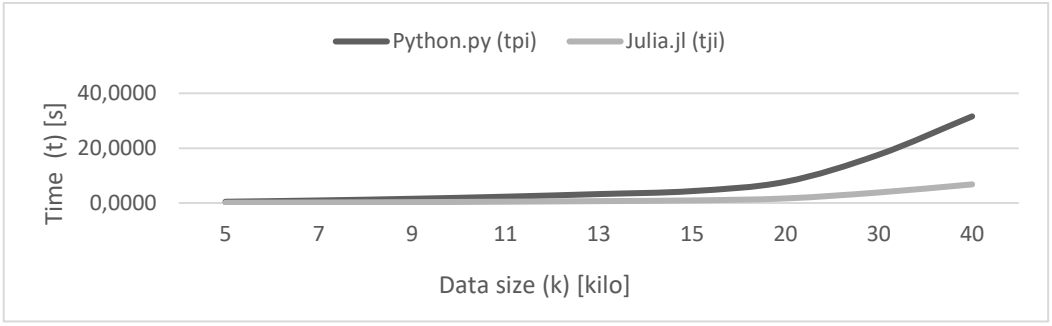

Fig. 1. Graphical representation of the performance result of insertion sort algorithm

Tab. 2. Summary of the results for bubble sort algorithm

Data size (k)	Execution time [s]		Julia / Python [%]	Python / Julia
	Python t_{pb}	Julia t_{jb}		
5000	1,8048	0,4291	24	4,2
7000	3,5627	0,8684	24	4,1
9000	5,9467	1,4296	24	4,2
11000	8,9204	2,1555	24	4,1
13000	12,6154	3,0343	24	4,2
15000	16,9405	4,0952	24	4,1
20000	30,2065	7,2517	24	4,2
30000	64,0401	16,2550	25	3,9
40000	124,0401	29,5770	24	4,2

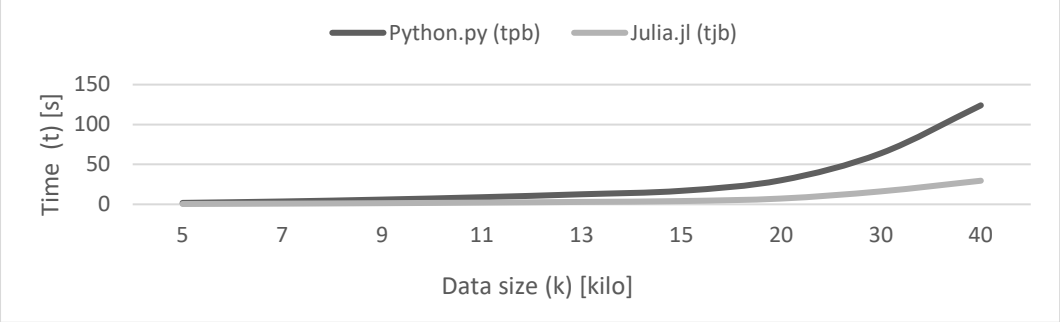

Fig. 2. Graphical representation of the performance result of bubble sort algorithm

Tab. 3. Summary of the results for quicksort algorithm

Data size (k)	Execution time [s]		Julia / Python [%]	Python / Julia
	Python t_{pq}	Julia t_{jq}		
5000	0,0113	0,0108	96	1,4
7000	0,0136	0,0074	54	1,8
9000	0,0160	0,0044	28	3,6
11000	0,0150	0,0083	55	1,8
13000	0,0237	0,0090	38	2,6
15000	0,0201	0,0098	49	2,1
20000	0,0311	0,0121	39	2,6
30000	0,0422	0,0159	38	2,7
40000	0,0632	0,0147	23	4,3

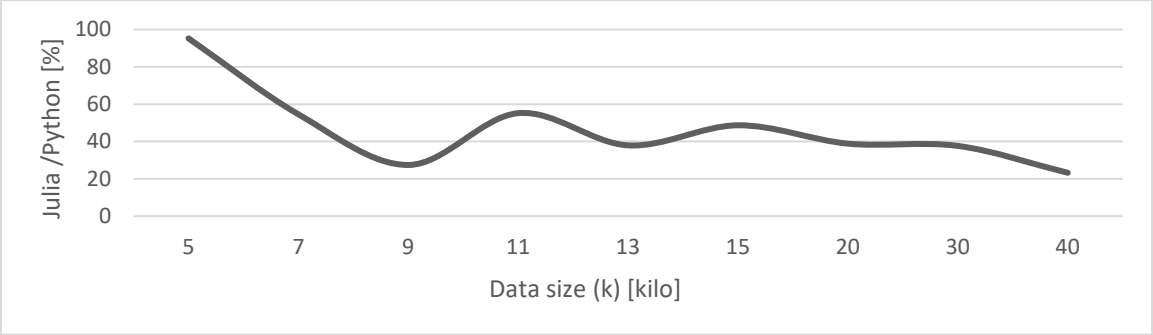

Fig. 3. Graphical representation of the percentage breakdown of results for both languages

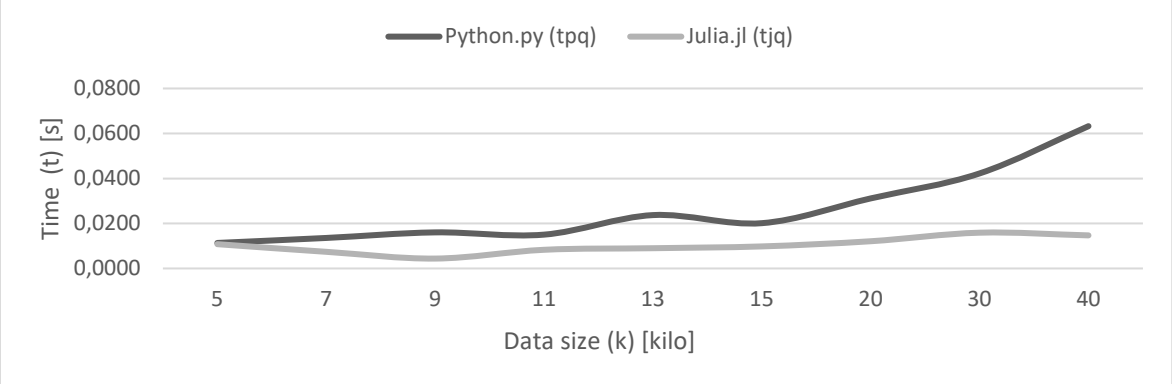

Fig. 4. Graphical representation of the performance result of quicksort algorithm

For sorting the largest number of elements, the difference in the execution time of the algorithms was as much as 124 seconds, where the Julia language alone solved the given processes in less than 30 seconds. The percentage difference is visible in all the conducted tests.

The results presented in the Table 2 can be expressed by models using the least-squares method and expressed by the formulas.

$$t_{jb} = (1{,}84 \cdot 10^{-8}) \cdot k^2 \quad (3)$$

$$t_{pb} = (7{,}59 \cdot 10^{-8}) \cdot k^2 \quad (4)$$

Appropriately for the Julia (3) and Python (4) languages. As in the previous case, both models being power functions achieved similar performance in data forecasting.

An increase in the value of a power for the model describing the values obtained through the Python language means that the values will grow faster, but it is worth noting that this value is not large. However, it indicates that as the number of iterations increases, the time for Python increases faster than for Julia. The results

of the research on the bubble sort algorithm were presented in the same way as for the other algorithms that sort the percentage results illustrate the change in the execution time of the algorithms written in Julia and Python.

The results presented in Table 3 clearly indicate that the code execution time is more beneficial for the Julia language, which on average needed 40% of the time needed by the algorithm in Python to perform the quick sort. Analyzing the data presented in the graph Figure 3, one may get the impression that they are unstable (it is difficult to draw a downward trend), presenting the result of the division expressed as a percentage. Where it is needed to point that the prepared data sets are random numbers, so it has impact on sorting.

The results presented in Table 3 can be expressed using the least-squares method as models and by mathematical formulas appropriately for the Julia (5) and Python (6) languages. Function t_{jq} (Julia) is a log-linear equation (using natural logarithms), whereas function t_{pq} (Python) is a linear equation.

$$t_{jq} = (1{,}09 \cdot 10^{-3}) \cdot \ln(\mathrm{k}) \quad (5)$$

$$t_{pq} = (1{,}54 \cdot 10^{-6}) \cdot \mathrm{k} \quad (6)$$

The difference between the mathematical models is definitely for quick sort. The model seen for the Julia language is almost linear when Python obtained a much weaker power relationship. Worth to note that the model for the Julia language obtained only a 60% fit to the data and is closely linear.

The graph presented in Figure 4 illustrates the advantage of the program prepared in Julia. The trend expressed for the algorithm written in Julia compared to the program written in Python increases slightly and the increase in the amount of data to be sorted does not cause any major problems. The Python file illustrates the increase in execution time with the increase of data set to be sorted, which resembles the graphs presented in the previous summaries (for the other sorting algorithms).

Tab. 4. Summary of results of tests conducted on GPU

Matrix size	**Execution time [s]**			
	CuPy.py	**TensorFlow.py**	**PyTorch.py**	**CUDA.jl**
10192	28,52	25,60	28,97	29,26
10192	28,54	25,62	28,75	28,96
10192	28,52	25,63	28,80	28,95
10192	28,55	25,60	28,86	28,93
10192	28,52	25,69	28,78	28,93
10192	28,55	25,63	28,80	28,94
10192	28,49	25,78	28,81	28,92
10192	28,55	25,70	28,82	28,92
10192	28,53	25,71	28,81	28,92
10192	28,51	25,65	28,86	28,91
512	0,10	0,06	0,10	0,06
512	0,16	0,03	0,21	0,04
1024	0,05	0,03	0,23	0,06
1024	0,04	0,03	0,04	0,05
2048	0,43	0,22	0,50	0,44
2048	0,51	0,22	0,51	0,50
2048	0,51	0,22	0,50	0,51
4096	2,05	1,68	2,07	2,04
4096	2,08	1,68	2,08	2,06
4096	2,08	1,69	2,08	2,08

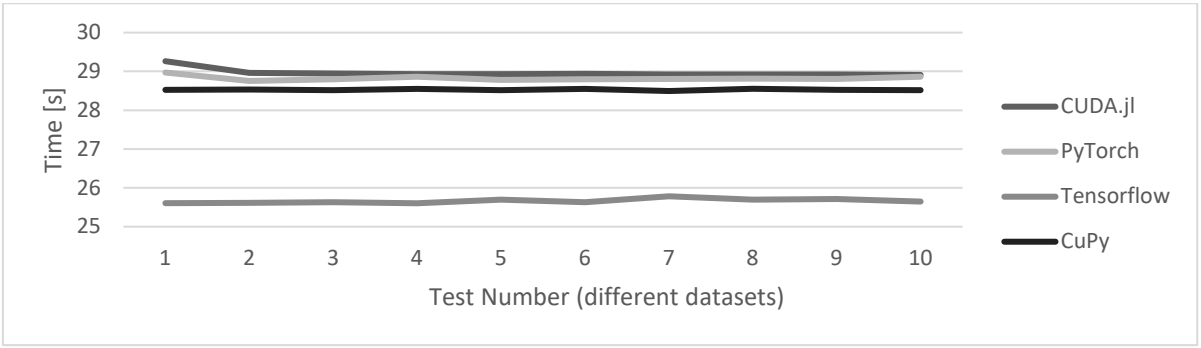

Fig. 5. Graphical representation of GPU results for large matrix operations

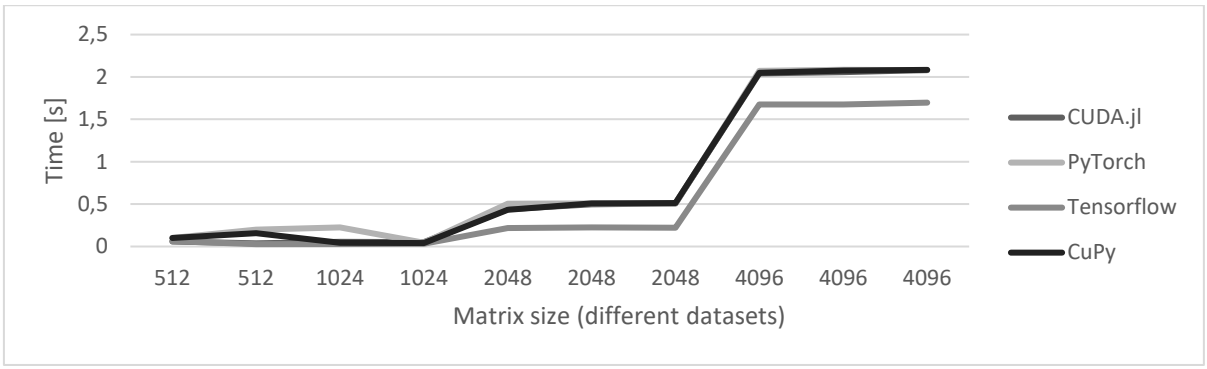

Fig. 6. Graphical representation of GPU results for small matrix operations

The tests conducted on the GPU were performed using the following software versions: Julia 1.9.4, Python 3.10.7, cuDNN 9.6.0 and VS Code 1.96.0. The computer hardware was not changed (the computing environment described in Table 1 was used).

The results of the tests conducted on GPU have been presented in tabular form (Table 4). All tests were repeated ten times, and the results are presented as their averages. The results obtained by the CuPy and PyTorch libraries are similar. These libraries encountered the greatest difficulties with the same data blocks. Both libraries obtained results similar to the results obtained from the execution of the code by the CUDA.jl library or were slower than it. The library that came out definitely the best is TensorFlow. It obtained the best results for each of the input sets. This library performed tasks on average 3 to 4 seconds faster than the CUDA.jl library. It is worth remembering, however that the TensorFlow library uses the optimized cuDNN library provided by Nvidia for computing on GPU. The cuDNN library is written in low-level languages, which significantly increases its computational capabilities and indirectly also the TensorFlow library.

Figure 5, which contains the results for large matrix operations and shows ten different results for various data sets. The chart behaves as expected (individual measurements do not differ significantly from one other). The execution time of the algorithms depends more on the amount of data than on its variety.

Figure 6 depicts how execution time scales with matrix dimension. Two independent runs were performed on for the 512 x 512 and 1024 x 1024 cases, whereas three runs were carried out for the 2048 x 2048 and 4096 x 4096 cases. In every described run matrices shared the same dimensions but were filled with different input datasets. Measurements obtained from runs that used different input data but the same matrix dimensions did not differ significantly.

The graphs (Figure 5 and Figure 6) indirectly show that the library implemented in Julia coped better with smaller matrices that required less memory allocation. This may indicate issues with processing data into an array format that is understandable by the GPU. When operating on large matrices, CUDA.jl coped definitely the worst of the previously described libraries.

3. Summary

3.1. Summary of research results

In the summary, the research findings for each of the presented algorithms are divided into a discussion of the results obtained during the tests on the CPU and GPU. In order to improve the readability of the data, all results for each sorting algorithm were presented in tabular form (Table 5) as values normalized to the results of the algorithm written in Julia for each of the sorts.

The results presented in Table 5 clearly shows that the Julia language is faster than Python. For the bubble sort and insertion sort algorithms, the difference between the execution of each of the algorithms was from four to five times. The results for quick sort are also unambiguous, but for small numerical values the difference was small. The execution time of the algorithms in both languages is dependent of the increase in the number of iterations performed during the tests. The more iterations were performed, the greater the difference between the results became.

Tab. 5. Summary of normalized all sorting results

Data size (k)	Julia / Python [%]		
	Quicksort	Bubble sort	Insertion sort
5000	96	24	22
7000	54	24	21
9000	28	24	21
11000	55	24	21
13000	38	24	21
15000	49	24	22
20000	39	24	21
30000	38	25	22
40000	23	24	22

Taking everything into account, models fitted with the least-squares method indicate that Julia is substantially faster than Python on a CPU, enabling significant savings in computational resources. This point is of paramount importance in the context of environmental protection.

Table 6 presents the results of the tests conducted on GPU for better data visualization they were presented in tabular form. The time results were normalized to the results obtained by the CUDA.jl library. Its analysis shows that the CuPy and PyTorch libraries written in Python performed slightly better or as well as the algorithm implemented in Julia based on the CUDA library. The TensorFlow library performed best of all the tested libraries, probably due to the use of the cuDNN library provided by Nvidia, which supports GPU calculations and is necessary to perform GPU calculations using TensorFlow. The cuDNN library is written in low-level languages which has a big impact on code performance.

The difference in this case is noticeable and for large matrices is about 11%, but for small matrices it is larger and sometimes reaches 50%. It is worth remembering that this cannot be a measurement error because the presented results are average values from 10 independent measurements. However, this difference decreases as the problem size increases, which may mean the previously mentioned problems with creating arrays that are understandable to the GPU via the CUDA library.

3.2. Conclusion and direction of further research

This research shows that algorithms written in Python languages achieve better time results than the algorithm using the CUDA.jl package, which is reflected in the time results. Visible differences appeared only in relation to the TensorFlow library, which, however has interfaces to a library written in low-level languages, which is why the obtained results do not allow for a clear answer to the question of which language performs better when using the GPU. However, it should be remembered that, as presented in [18]. The execution time of each language is affected by the type of problem. Julia, during the tests conducted on the CPU [8], [9], [18] obtained different results (it did minimally better or worse than Python), which was influenced by the way the matrix multiplication operation was implemented. Therefore, a broader analysis of the temporal performance of each language when running on the GPU should be undertaken.

To validate the study's findings a more in--depth analysis of how each library loads data into GPU memory should be carried out and further experiments should be performed on additional GPU operations. Further research should focus on comparing more complex operations performed on the GPU. For example, it is worth checking how the use of the GPU allows for accelerating the training of models used in developing artificial intelligence or machine learning. It is also worth checking how programming languages operate on graphics cards from manufacturers other than NVIDIA. As well as on other operating systems. On the other hand, it would be worthwhile to investigate how Julia performs when running its own kernels on CUDA.

Tab. 6. Summary of test results performed on GPU normalized to Julia

Matrix size	CuPy.py / CUDA.jl	TensorFlow.py / CUDA.jl	PyTorch.py / CUDA.jl
10192	0,97	0,87	0,99
10192	0,99	0,88	0,99
10192	0,99	0,89	0,99
10192	0,99	0,89	1,00
10192	0,99	0,89	0,99
10192	0,99	0,89	1,00
10192	0,99	0,89	1,00
10192	0,99	0,89	1,00
10192	0,99	0,89	1,00
10192	0,99	0,89	1,00
512	1,74	1,10	1,77
512	4,26	0,75	5,57
1024	0,82	0,55	4,12
1024	0,83	0,69	0,91
2048	0,97	0,49	1,13
2048	1,01	0,45	1,02
2048	1,01	0,44	1,00
4096	1,00	0,82	1,01
4096	1,01	0,81	1,01
4096	1,00	0,81	1,00

List of commonly used abbreviations and terms

.jl	Extension of programs written in the Julia language.
.py	Extension of programs written in Python language.
Julia	Julia programming language reference.
Python	Python programming language reference.
Algorithm	It is a set of instructions or logical rules specifying a sequence of operations to be performed by a computer system to solve a specific problem.
Execution time	The time required for a computer system or program to perform a specific task or operation. Execution time is measured in various units.
Development environment	A set of tools, libraries and other resources that are required to run and execute a computer program.
Editor	A computer program used to create, edit, and format text. In the context of computer science, a text editor is used to write source code for computer programs.
Compiler	It is a computer program that converts source code written in a programming language into equivalent machine code or intermediate code executable by the computer environment. This process is called compilation.
CPU	Central Processing Unit is the main processor of a computer, responsible for executing general-purpose tasks, managing system operations, and handling sequential processing efficiently.
GPU	Graphics Processing Unit is a specialized processor designed to handle parallel computations, primarily for rendering graphics and accelerating tasks like machine learning and scientific simulations.
CUDA	Parallel computing platform and API developed by NVIDIA that enables developers to harness GPU power for high-performance computing. It significantly accelerates tasks like simulations, image processing, and machine learning.
RAM	Random access memory type of computer memory used to store working data and machine code.

4. Bibliography

[1] "General Python FAQ". 3.11.11 Documentation, Python Software Foundation (online: 04.03.2025).

[2] "Frequently Asked Questions", Julia Language Documentation, Franklin.org and others (online: 04.03.2025).

[3] "Julia programming language", University of Cincinnati Libraries, 2023 (online: 25.05.2025).

[4] White M., "The Need for Speed: Julia vs. Python", 2022 (online: 04.03.2025), DOI: 10.13140/RG.2.2.16468.32646.

[5] "Julia in Visual Studio Code", Microsoft (online: 04.03.2025).

[6] "Python in Visual Studio Code", Microsoft (online: 04.03.2025).

[7] "Learn To Code With Visual Studio Code", Microsoft (online: 04.03.2025).

[8] "JuliaLang/Microbenchmark", "GitHub", Franklin.org and others (online: 04.03.2025).

[9] "Benchmarks", Franklin.org and others (online: 04.03.2025).

[10] Knuth D.E., *Sztuka programowania*, WNT, Warszawa 2002, ISBN 83-204-2539-5.

[11] Strzałka D., Grabowski F., "Wybrane właściwości statystyczne dynamiki procesu sortowania przez wstawianie", *Metody Informatyki Stosowanej*, Tom 1(18), 85–98 (2009).

[12] Cormen T.H., et al., *Wprowadzenie do algorytmów*, Wydawnictwo Naukowe PWN, Warszawa 2012.

[13] "PyTorch", The Linux Foundation (online: 04.03.2025).

[14] "TensorFlow Home Page", Google (online: 04.03.2025).

[15] "CUDA programming in Julia", Julia language documentation, Franklin.org and others (online: 04.03.2025).

[16] "CuPy Home Page", Preferred Networks (online: 04.03.2025).

[17] "Julia language home page (Blog). Announcing the release of Julia 1.0", Franklin.org and others (online: 04.03.2025).

[18] Maj M., "Performance comparison of Julia and Python programming languages based on available literature", *Computer Science and Mathematical Modelling*, No. 20, 5–17 (2024).

Porównanie wydajności języków oprogramowania Julia i Python na GPU i CPU

M. MAJ

W pracy przedstawione zostały badania nad wydajnością czasową każdego z języków podczas wykonywania operacji na CPU (jednostce centralnej) i GPU (jednostce przetwarzania grafiki). Artykuł ten należy traktować jako spójną kontynuację części pierwszej [18]. Praca została podzielona ze względu na kryteria redakcyjne. Artykuł zawiera opisy wykorzystanych technologii oraz algorytmów, które posłużyły celom badawczym. Przedstawione zostały algorytmy przygotowane z dbałością o zachowanie jednakowego wykorzystania funkcji przez oba języki programowania na podstawie schematów blokowych oraz pseudokodów. W pracy najpierw wskazano badania przeprowadzone na podstawie algorytmów sortowania zaimplementowanych przez autora i wykonanych przez procesor. W kolejnej części artykułu znalazły się badania weryfikujące wydajność języków programowania Julia i Python na kartach graficznych, ze szczególnym uwzględnieniem implementacji operacji macierzowych. Artykuł został zakończony omówieniem wydajności czasowych każdego języka programowania.

Słowa kluczowe: Python, Julia, porównanie, wydajność.