ETH zürich

Performance comparison of high-level programming languages:

Julia, Python and Rust

by Michelle Schneider, Paul Ochs, Vsevolod Semenov

August 31, 2023

Abstract

In this study we evaluate the high-level programming languages Python, Julia, and Rust in the context of scientific computing and numerical analysis. Our aim is to balance programmer efficiency with optimal execution speed. The research employs a fourth-order numerical diffusion equation as the benchmark algorithm. While Python provides versatile implementations including CuPy; Rust and in particular Julia demonstrate superior execution speed. Rust offers fast performance and high memory safety, making it a strong candidate for tasks requiring efficient and secure parallel computing, but there is an initial learning curve. Julia, on the other hand, offers an intuitive, high-level syntax and seamless GPU compatibility and thus enhances productivity without compromising performance. These findings highlight the trade-offs in language selection. Ultimately, language choice depends on programmer knowledge, task specifications, and desired performance outcomes.

Contents

1 Introduction

In the rapidly evolving landscape of scientific computing and numerical analysis, the quest for achieving both high programmer productivity and optimal performance remains an ongoing challenge. While Python's NumPy library offers a variety of easy-to-use solutions that facilitate project development, by its nature Python as a very high-level, interpreted language appears quite slow when compared to faster low-level languages such as C or Fortran. These faster languages offer great performance but require deeper understanding and more verbose code during development. Languages like Julia and Rust, situated somewhere between very low-level and very high-level languages might be able to offer easier and faster development while not missing out on the performance side. This project aims to investigate this idea by comparing Julia, Rust and Python implementations.

2 Algorithm

The algorithm used to compare the performance of the different languages and implementations is the fourth-order numerical diffusion equation on a two-dimensional plane given by:

$$\frac{\partial \phi}{\partial t} = -\alpha_4 \Delta_h (\Delta_h \phi) \tag{1}$$

This is particularly interesting in the context of weather and climate models as this diffusion equation presents an essential component of the larger models. As with the larger models our simulations treat the vertical scale differently, in this case we have two-dimensional diffusion. This topic also holds great importance in the field of high-performance as demonstrated during the lecture. Regions of heavy computations often present bottlenecks and are thus the place where performance matters especially. The finite-difference discretization gives rise to the following stencil

$$\Delta_n \phi_{i,j}^n = (-4\phi_{i,j}^n + \phi_{i-1,j}^n + \phi_{i+1,j}^n + \phi_{i,j-1}^n + \phi_{i,j+1}^n)/\Delta x^2 \tag{2}$$

$$\partial_t \phi_{i,j}^n \approx (\phi_{i,j}^{n+1} - \phi_{i,j}^n)/\Delta t \tag{3}$$

For the full reasoning and explanation see Slides 02 [1]. With $\alpha = \alpha_4 \frac{\Delta t}{\Delta x^2}$ this results in the following algorithm:

```
do num_iter:
    update_halo(in)
    tmp = lap(in)
    out = lap(tmp)
    out = in − $\alpha$ out
    swap(in, out)
```

3 Implementations

The above described algorithm is implemented in Julia and Rust. The Python implementations from the lecture are used as comparison. For each language, Julia and Rust respectively, a sequential "naive" version, vectorized single threaded, and multithreaded versions are implemented for the sake of this project.

3.1 Code Organization

The code for this project is organized in the following manner: For every language there is a folder where the source code can be found in the `src/` directory. All algorithm parts come with the `stencil2d`-prefix. There is also always a main-file (`JuliaCode.jl`, `benches.py` and `main.rs`) which run the respective benchmarks. Furthermore there are some helper files like the `input_loader`, that help with generic functionality which does not necessarily belongs into the main file. For Julia and Rust some basic testing infrastructure is also available. All of the code can be found in the Gitlab repository [2].

3.2 Python

This is our base case, for which the following implementations are available: for-loop naive, NumPy, MPI and CuPy. The code is essentially the same one that we got during the HPC4WC course, with slight modifications.
The naive implementation uses pure Python for-loops. It is known that pure Python is especially slow and in normal use cases, libraries like NumPy would be the advised Python approach. Thus this naive for-loop is mainly just for comparison with the other for-loop implementations. The next version is the NumPy implementation presented in the lecture. This uses slicing operations on NumPy arrays.
Finally the MPI and CuPy versions presented in the lecture were used to compare to the multithreaded and GPU versions.

3.3 Julia

The Julia language is a newcomer among the HPC languages. The motto of this language is: "Looks like Python, runs like C". In this report we test that hypothesis. But why should it even be possible? How could Julia achieve this? The main ingredient in the language is using a just-in-time (JIT) compiler, which translates the code written by the programmer into machine code, as soon as the file is executed. While this results in a noticeable overhead, it is usually a one-time occurrence as long as the terminal/REPL remains open, since Julia caches all the compiled functions.
For the sake of this report, there are five implementations available: naive, for-vectorized, multithreaded, "hand"-vectorized and CUDA.
The naive version essentially consists of simple for-loops while considering the standard column-major array layout of Julia. This means that the inner-most loop iterates over the first index of an array. Note that the code's appearance very much resembles Python, but a one-to-one

translation is quite tedious as Julia is 1-index-based and `numpy` is in row-major. The functions `laplacian`, `update_halo` and `apply_diffusion` look exactly like one would expect after seeing the naive Python implementation.
In the spirit of "never do yourself what others can do for you", we were able to easily implement the for-vectorized version. There is a Julia package called `LoopVectorization.jl`[3] and it does exactly what the name suggests: vectorizing loops. While the inner workings of this package are advanced topics, all the user needs to do is to apply the `@turbo` macro to the outermost loop of an array update. In our case it is always the one over the z-axis. The code looks like this: `@turbo for k in 1:nz`, followed by the loop body. We made use of this in every function for this version.
Seeing how remarkably easy it was to vectorize a loop, one might wonder if something similar exists for parallelization. Luckily - for this specific case - we did not need to look very far! The same package also allows for automatic multi-threading by simply replacing `@turbo` by `@turbo thread=true` to use all the available cores (usually half of all available threads) or specify an exact number (e.g., `@turbo thread=4`) to use a desired number of threads for that loop-body. Since we usually want to use all the available cores anyway and typing is cumbersome, there is a convenient short-hand notation for that: `@tturbo`. This is the macro that we used.
So far we only talked about our naive version and two slight variations of it (only six more words per file) and nothing Julia-specific (except for the macros). What about actual coding and refactoring? We implemented a "hand"-vectorized version of the code. Optically, it resembles Python's Numpy-version very much. The idea is exactly the same: Get rid of all the for-loops and replace them with array-expressions. Julia does not rely on external libraries to provide the same functionality. However, there are some caveats while directly translating from Numpy: First of all, it is essential to know about broadcasting. A broadcast is nothing else than an element-wise operation where the order of execution does not matter. A prime example for this is array-addition, where we only want the correct elements to be added together, in which order does not matter. The benefit of this is giving the compiler additional freedom to optimize the code. Broadcasts should always happen when an operation or function of scalars is applied to a collection of scalars. To achieve this, one merely needs to add a dot before the operator/function parentheses: `arr1 .+ arr2` or `sin.(arr3)` or using the `@.` macro for longer expressions: `res = @. arr1 + arr2 / arr3`. Of course, the array dimensions must match, otherwise an error is thrown, which brings us to the next topic: bounds-checking. Whilst writing and debugging code it is helpful having an error thrown instead of a complete crash. For deployment, however, we might want to get rid of these bounds-checks for the sake of a little bit of extra performance. This can be accomplished by using the `@inbounds` macro. Finally, when working with arrays we want to avoid unnecessary copies as much as possible. To achieve this, we can create views which are essentially references/pointers to some memory location inside a given array. The `@views` macro allows us to do exactly that for entire array-expressions and thus saves us both time and memory.
The final version we want to present - the GPU accelerated CUDA version - is the most interesting one. Why? There is none. For this simple, array-based task we did not even need

to copy paste code nor write custom GPU kernels! All we needed to do is: Having an NVidia GPU and the CUDA Toolkit installed, importing the `CUDA.jl` package[4] and pre-allocating the arrays in GPU memory via the `CUDA.cu` function instead of regular RAM for our "hand"-vectorized example from above. Since there are no loops except for the time iteration and no new explicitly-typed variables (we always work with the given types) it simply works! Please note that in comparison to the CPU computations which default to `Float64` our GPU is only capable of `Float32` computations. Thus, the results might be less accurate. Unfortunately, none of the other versions run directly on the GPU, therefore "hand"-vectorization is required.

3.4 Rust

Like Julia, Rust is quite a young and recent language with the first stable release being less than 10 years old. Rust focuses strongly on performance, type and memory safety and concurrency, which makes it very attractive for systems programming (only language next to C and assembly with Linux kernel support [5]). These features also make Rust interesting as an HPC language, which is investigated in this project. In total four different versions of the stencil computation were implemented in Rust: a naive version, two vectorized and one multithreaded version.
As the stencil computation essentially consists of looping over arrays, in this case 2D and 3D arrays, the different versions boil down to different ways of iterating. Throughout the Rust implementation, the Rust crate `ndarray` [6] is used to save the data. This library allows easy handling of multidimensional arrays as well as mathematical operations on those arrays. The different versions of the stencil computation implement the Laplacian computation. Initially also the halo update was implemented in different versions. However, in the benchmarks only the naive halo update was used. The reason for this is that with some version, the suspicion arose that the halo update of a specific version might be relatively more inefficient to a point that it over-shadows the actual Laplacian computations (e.g. because of inefficient memory usage). With more experience in Rust, faster halo updates should definitely be possible.
The base or naive version implements the halo update using only native Rust for-loops. A feature of `ndarray` is the option to set the memory layout either to row-major or column-major by the user. Setting the layout to column-major can be achieved by appending `.f()` to the shape input when initializing an array. Having this option meant that the Fortran code could be used as a close example to base this version on, although all indices needed to be shifted as Rust is 0-index based.
Next, different versions using `ndarray` library functions were implemented. Notably, the code called `stencil2d_vectorized.rs` uses mutable and immutable slices over indices of the arrays to compute the Laplacians. This version resembles the NumPy code from the lecture the most. In this version the `update_halo` function presented some difficulties. Slicing on the same array once in a mutable slice (the halo region where the values are being copied to) and once in a immutable slice (interior region where values are copied from) was not easily achieved in Rust. Indeed, with our current level of experience some sort of temporary arrays were necessary, which resulted in more copy statements.
Another version using `ndarray` functions called `stencil2d_iterators.rs` aims to implement

the NumPy version in a slightly more Rust friendly way. The `ndarray` crate provides a very convenient macro `azip!`, which allows iterating over elements in arrays directly, while in turn allowing iteration over several arrays or slices. This means that a zip operation could be used to iterate over slices of the arrays and calculate the Laplacians from the elements directly. Within this `azip!` statement we once iterate over a mutable slice of the result field and five times (once for the current cell and then four cells around the current cell according to the stencil) we iterate over an immutable slice of the input field.
Finally a multithreaded version was implemented using the Rust crate `rayon`[7]. The two library `ndarray` provides the macro `par_azip!` which performs the same operations as the single threaded version but distributed over several threads using the library `rayon` in the background.

4 Benchmarking

To asses the performance between the different implementations and languages we performed a series of benchmarks. While keeping the total number of iterations constant and set to 100, the problem sizes were varied. All of the presented languages feed on the same input.

4.1 Python

The Python benchmarks were performed using the `time` package. While this lacks any sophisticated statistical analysis and purely allows to time a region of code, it is very easy to use. We collected ten samples and then took the one with the fastest run-time. The idea here is to have a low estimate, since many interrupts happen during the benchmark on a daily used system and we wanted to approximate the run-time on a dedicated cluster.

4.2 Julia

There exists a very useful package for benchmarking in Julia called `BenchmarkTools.jl`[8]. We implemented our benchmarks via the `@belapsed` macro, which returns the fastest run out of a series, if the benchmarked function is quick (less than a couple of seconds). There is some autotuning going on in the background which remains hidden from the user. The macro provides a `setup` option, where we (re)allocate our initial arrays for each run. For the CUDA version, we additionally made use of the `CUDA.@sync` macro to ensure that we actually measure the computation and not only the GPU-call. Please note that we also (re)allocate our initial arrays on the GPU, therefore we do not measure the overhead of copying data.

4.3 Rust

The benchmarks of the Rust implementations were performed using the very nice benchmarking crate `criterion` [9]. Not only does this allow the user to quickly and easily set up sensible benchmarks, but the library also directly provides a large number of statistical analyses and plots. This allows for much more insight into the actual runtimes and distribution of runtimes than just timing the code. Ultimately, it was decided to use the lower bound of the confidence interval around the median of the measurements as the benchmark time for a specific version. As already touched upon above, the idea here was to try to exclude other processes in the machine that might slow down the execution needlessly. Note that the criterion itself provides a best estimate for the run time, as well as mean and median and confidence intervals and standard deviations. Since the Python benchmarks were not able to provide any such statistical analysis we also omitted these for the most part here.

5 Discussion

5.1 Comparison

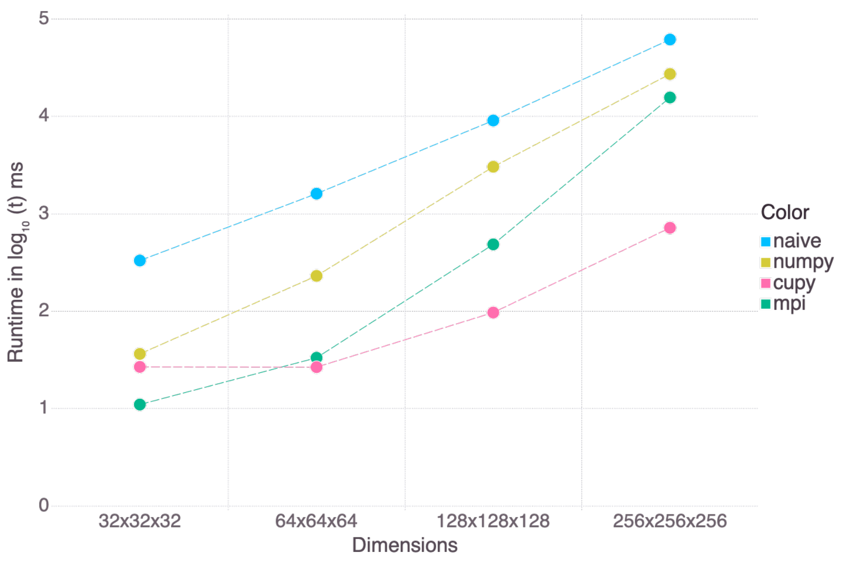

Figure 1: Benchmarks of the different Python implementations. The "naive" version uses pure Python for-loops, "numpy" uses slicing over NumPy arrays, "cupy" was run on NVidia GPUs and uses the CuPy library. Finally "mpi" was run using MPI and was run on 16 cores.

Figure 1 shows the results obtained from running the Python benchmarks. As all of these implementations, except the naive one, were already discussed in the lecture, we knew what to expect here. It is worthwhile to note how fast the CuPy version is, even though it is still Python and nothing was done except running with the array on a GPU. This again highlights the great potential of GPUs in HPC.
In Fig. 2 we see the results from the Julia benchmarks. Due to the advantage of being a compiled language, the naive version already outperforms Python's `numpy` implementation. Apparently, the JIT-compiler is able to further optimize some of the given operations in comparison to the ahead-of-time compiled `numpy` package. It is noteworthy that the compilation time is excluded from the benchmark timings. Looking almost identical- but slightly better - is the vectorized version. We assume that the runtime improvement comes from omitting bounds-checking and for-loops. The for-vectorized version is the first one that does not look perfectly linear: It is almost an order of magnitude faster for the two smaller data sets and just slightly better for the two larger ones. We believe this has to do with caches, where the operations can be executed quicker while the information is still in the L3 cache. However, it is surprising that the change happens from the second to the third data set, as there are 32MB of L3 cache per core cortex.

This might be because other programs or the code uses a substantial amount of said cache. For the multithreaded version we first encounter a regression in performance due to the threading overhead. Here, the jump happens where we expected it, maybe because the combined 64MB L3 cache is enough to store both the array data and the rest. This version is also faster than the naive or "hand"-vectorized approach in every case. Finally, the GPU version beats everything by around an order of magnitude for the larger data sets. This is not surprising, as the operations in the algorithm can be executed in an embarrassingly parallel fashion. This implementation also beats `CuPy`, most likely due to the smaller calling-overhead and slightly better optimization. Is this the best what can be achieved? Probably not. Manually optimizing this code for L1, L2 and L3 caches would most likely yield even better results, but this is outside of the scope of this project.

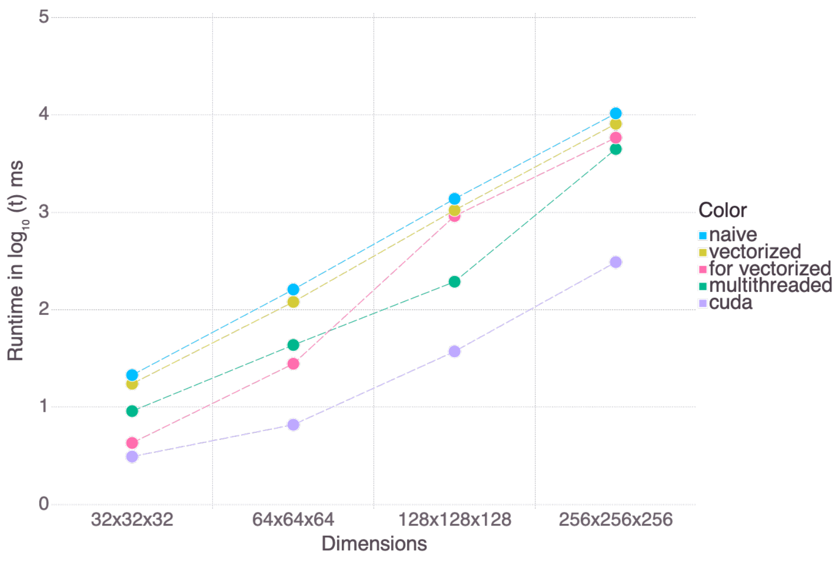

Figure 2: Runtimes obtained from the Julia benchmarks. The "naive" version uses only for-loops. The "vectorized" version uses the `@turbo` macro from the `LoopVectorization.jl` package for automated vectorization. "Multithreaded" used automated multi-threading with the `@tturbo` macro. Finally, the "cuda" implementation just used the `CUDA.cu` function on a "hand"-vectorized version and ran it on NVidia GPUs.

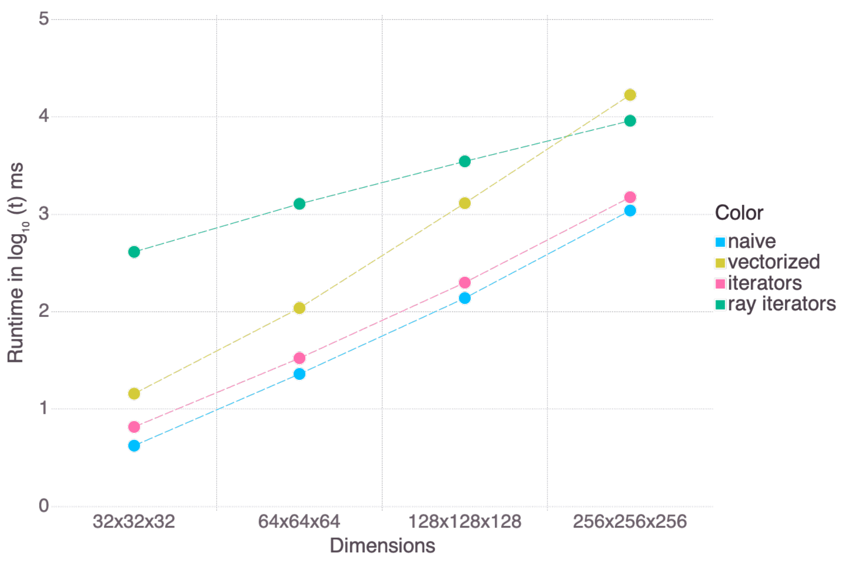

Figure 3: Runtimes obtained from the Rust benchmarks. "Naive", denotes pure Rust for-loops, "vectorized" denotes slicing simliar to NumPy, "iterators" denotes the version using the `ndarray` macro `azip!` and "ray iterators" the same macro but using `rayon`.

Looking at Fig. 3, which shows the benchmarking results for our Rust implementations, we see a few curious things. Most notably we see that the naive implementation performs the fastest, just slightly faster than the iterator version with `azip!` macro. This came as a bit of a surprise, we had expected another version, one that does not just use naive for-loops to be the fastest. Due to time constraints and not enough knowledge and experience with the language, we are not entirely sure what is happening but we can make some educated guesses. Most of those guesses boil down to the compiler being very good at optimizing.

A first thing to note is that one reason we thought why the naive version might run slower is because index bound checks are likely performed on every element access. It might be possible that the compiler unrolled the loops and optimized away unnecessary bound checks. If this does make a difference, we would have assumed similar optimizations being possible in the iterator version. Since `azip!` is a library macro, we would also have assumed this implementation to be more efficient than naive for-loops. An unrolled version was briefly tested but it performed worse than the naive, further hinting at very good compiler optimization. It was also investigated whether passing the arrays as mutable references or mutable references to mutable views had an effect, but no significant difference was observed.

Another reason for the unexpected behavior might be related to the problem size and the computational load. It may be that the relative runtimes we are seeing change for larger problem sizes, as relative overheads become smaller and the amount of computation to be done increases. This might especially affect the parallel implementation. If the overhead of spawning threads is too large compared to the amount of computations per thread there might not be any gain from running the code multithreaded. Looking at a profiler, at least for smaller problem sizes, there did seem to be a lot of time being spent on processes related to handling the threads compared to calls to `apply_diffusion`. We also observed the interesting behavior that the performance of the multithreaded version increased if we built the benchmark executables with fat link-time

optimization (compared to the default thin), while all other versions got slower. After some research we found this might be related to the number of code generation units, but we did not investigate any further.
Finally a note on the vectorized version. This uses slicing and mathematical operations on the arrays directly, resembling NumPy code. This was the first version implemented after the naive one. We realized that this might all in all not be an efficient way to solve this problem and thus the iterators version was implemented. We assume that the slow runtime is due in large to inefficient memory layout and or accessing.

5.2 Exploring Julia: A Language Beginner's Perspective

The Julia code, naive version, was written by a team member with limited programming experience, mainly in Python, and without any experience in Julia. After getting used to the switch from 0-index-based to 1-index-based counting, the translation of the Python code felt more or less straightforward. The member did not notice any fundamental differences in syntax compared to Python. They were positively surprised by Julia's impressive performance and responsiveness, especially when dealing with such a computationally intensive task. It was also interesting to see that in Julia there was no need to use an additional package, like NumPy in Python, in order to deal with arrays. When trying to plot the results for validation, however, the member felt like there were less options than in Python. The library that was used (Gadfly) had beautiful visualizations, but in terms of customizability, the member still prefers Python's libraries, like Matplotlib. But this could very much also be attributed to habituality, which leads us to the general theme of this member's experience in learning Julia: On the surface, Julia feels so similar to Python that there is little curiosity to make the switch. However, as the member dived a bit deeper into Julia, and will most likely also continue to do so, a whole range of advantages compared to Python were discovered. This makes Julia a real contender to perhaps even be the standard programming language that the team member will use in the future.

5.3 Exploring Rust: A Language Beginner's Perspective

The Rust code was implemented by a team member without any prior experience in Rust but with a background mostly in C++ and Python. The initial steps involved familiarizing themselves with Rust's syntax, data structures, and memory management concepts. There are some similarities with C++ relating to pointers and references, however, there are also vast differences, mostly pertaining to Rust's memory model. The team member encountered challenges in adapting to Rust's ownership model and strict borrow checker, which requires a reevaluation of traditional programming practices. This becomes especially apparent when the Rust compiler does not allow more than one reference to some object or when some operations produced errors because of the mutability of the references involved. Because of this, the team member is not sure if the performance of a specific version is dominated by inherent limitations of the method (e.g. more checks than other operations) or if the observed performance is simply a result of sub-optimal or too much memory usage. The team member also feels that this might be one drawback that Rust has: Known ways of solving problems in other languages do not always work the same way in Rust and thus there is a steep learning curve at the beginning. Even though the team member considers themselves to still be within that curve, they did enjoy getting to know Rust a lot and could imagine using it in the future.

6 Conclusion

When we set out on this project, we wanted to investigate if it made sense to switch to Julia or Rust in an HPC environment and how those languages compare to other high (or not quite as low-level as C) languages such as Python. We can confirm that both Julia and Rust are able to run faster than most Python versions. This is not surprising, it is to be expected based on inherent language designs. A notable exception is Python's GPU version using CuPy which is able to be perform extremely well, comparable to the fastest Julia implementation using CUDA. From this we conclude and confirm a trend that is already happening, namely that GPU programming has great potential in HPC use cases.

With respect to Rust we see the great potential the language has: Being designed for memory safety could be beneficial for HPC and parallel computing. However, we also note the learning curve if no prior knowledge exists. This means that, although a fast and efficient solution might be attainable, it might not be easily achieved without more knowledge of the language. Another caveat of Rust is slow (or not yet completely stable) support for known HPC paradigms and GPU programming. Especially with GPU programming there exists a difficulty between Rust's memory model and that of the GPU.

This brings us to Julia, a language in similar regions of Rust in terms of performance but with a much higher level syntax. This often means faster development time and less of a learning curve. The large number of libraries that Julia has to offer, including multi-threading and GPU support, which make substantial performance improvement very simple, demonstrate the capabilities of Julia in HPC. If, however, the absolute maximum performance is required, a more sophisticated approach is needed.

References

[1] O. Fuhrer et. al. *HPC4WC*. https://github.com/ofuhrer/HPC4WC. 2023.

[2] Schneider Michelle, Ochs Paul, and Semenov Vsevolod. *Performance comparison of high-level programming languages: Julia, Python, Rust*. https://gitlab.ethz.ch/vsemenov/hpcwc-performance-comparison. 2023.

[3] E Lilly et. al. C. Elrod. *LoopVectorization.jl*. https://github.com/JuliaSIMD/LoopVectorization.jl. 2019.

[4] D. Lin et. al. *BenchmarkTools.jl*. https://github.com/JuliaGPU/CUDA.jl. 2013.

[5] Liam Proven. *Linux 6.1: Rust to hit mainline kernel*. https://www.theregister.com/2022/10/05/rust_kernel. October 2022.

[6] *Crate ndarray 0.15.6*. https://docs.rs/ndarray/0.15.6/ndarray/index.html. 2023.

[7] *Crate rayon 1.7.0*. https://docs.rs/rayon/1.7.0/rayon/index.html. 2023.

[8] J. Revels et. al. *BenchmarkTools.jl*. https://github.com/JuliaCI/BenchmarkTools.jl/tree/master. 2015.

[9] *Crate criterion 0.5.1*. https://docs.rs/criterion/0.5.1/criterion/index.html. 2023.

A Benchmark Environment

All of the benchmarks are conducted on a local machine running Manjaro XFCE (arch-based) on the Linux kernel 6.4.9-1. The specifications of the machine are the following:

CPU	AMD Ryzen 3950X 16C/32T @ 3.4GHz
GPU	NVidia RTX 2080 Super
RAM	32GB @ 3600MHz

Table 1: System Specifications

B Languages, Packages and Versions

We use the following versions of the languages for the code and the benchmarks:

Julia	1.9.3
Python	3.11.3
Rust	1.70

Table 2: Languages and Versions

The packages used in every language are:

Julia	BenchmarkTools, CSV, CUDA, DataFrames, Gadfly, LoopVectorization
Python	Numpy, Pandas, CuPy, MPI4PY, Time
Rust	criterion, csv, ndarray, rayon

Table 3: Packages