ETH Zurich

Institute of Atmospheric and Climate Science

Writing a "Modern" Climate Model Using a DSL

Nicolai Krieger ⟨nkrieger@student.ethz.ch⟩
Nadja Omanovic ⟨onadja@student.ethz.ch⟩
Vera Schönenberger ⟨verasc@student.ethz.ch⟩

Date: 04-08-2020
HPC4WC - Report
Supervisor: Oliver Fuhrer

1 Introduction

Fortran is still one of the most used programming languages in climate modeling and provides fast handling of computationally expensive calculations. However, Python is getting more popular in the scientific community, and the question arose if a climate model can be efficiently run based on Python. This project aims to compare the performances of Fortran and Python for the sea ice parameterization of the FV3GFS climate model. A GT4Py (GridTools for Python) structure was adapted to include different backends and analyze their respective performances.

Sea ice is strongly influenced by the atmosphere and the ocean. Changes in sea ice extent, thickness, and concentration affect the atmospheric and oceanic conditions. A higher sea ice content results in a higher albedo, i.e. a higher reflection of incoming solar radiation. This enhances the sea ice formation, as the surface is cooled. Hence, a positive feedback mechanism exists. In addition, a sea ice cover impedes the heat and water exchange between the atmosphere and ocean, causing smaller surface fluxes. This can lead to major changes in the density structure of the water directly affecting the oceanic circulation.

The sea ice scheme in the GFS model is based on three-layer thermodynamics predicting sea ice thickness, surface temperature, and ice temperature (Winton, 2000). The model has four prognostic variables: snow layer thickness, ice layer thickness, upper and lower ice layer temperatures. The lower ice layer temperature is fixed at the freezing temperature of seawater, while the upper ice layer temperature is determined from the surface energy balance. The ice temperature serves as a basis to calculate the changes in energy fluxes, ice, and snow mass, consequently.

The scope of this project includes porting code from Fortran to Python and implementing a GT4Py structure to run the parameterization on CPUs and GPUs. GT4Py is a domain-specific language (DSL) for stencil operations. It is embedded in Python, and therefore the Python syntax is applied. In GT4Py, stencils are compiled using different backends. Depending on the chosen backend, the stencil is compiled using different programming languages. It is possible to run the stencil script on GPUs with the backend *gtcuda*.

In this report, performance tests for the backends *numpy*, *gtx86*, and *gtcuda* will be conducted and compared to the performance of Fortran. For the backend *numpy*, the stencil is computed using vectorized Python syntax, while C++ is used, when using the backends *gtx86* and *gtcuda*. As mentioned before, using *gtcuda* enables running the stencil on a GPU, whereas a CPU is used for the backends *numpy* and *gtx86*. In the following, the results from the performance analysis are shown and discussed in Section 2. The generated scripts are available on GitHub[1].

[1] https://github.com/VulcanClimateModeling/physics_standalone/tree/GT4Py/seaice

2 Results

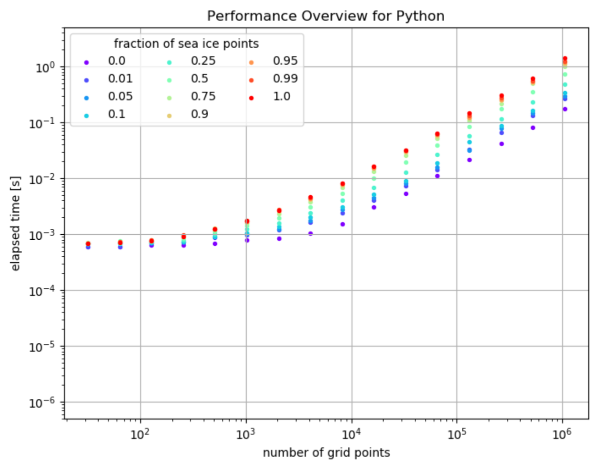
In this report, the sea ice parameterization of FV3GFS (c48 simulation) was used to test the performance of Fortran, Python, and the different backends of GT4Py. Since the sea ice parameterization can be written in a stencil structure, it is well suited to test the performance of the different GT4Py backends. For all the different test cases, the sea ice parameterization was run for a varying number of grid points (2^5, 2^6, 2^7, 2^8, 2^9, 2^{10}, 2^{11}, 2^{12}, 2^{13}, 2^{14}, 2^{15}, 2^{16}, 2^{17}, 2^{18}, 2^{19}, and 2^{20}) and different fractions of grid points with sea ice (0 %, 1 %, 5 %, 10 %, 25 %, 50 %, 75 %, 90 %, 95 %, 99 %, and 100 %).

Ten iterations were performed for each possible combination of different grid size and sea ice fraction to get a more robust result. The medians of these iterations for each combination of grid size and sea ice fraction are displayed in Figure 1.

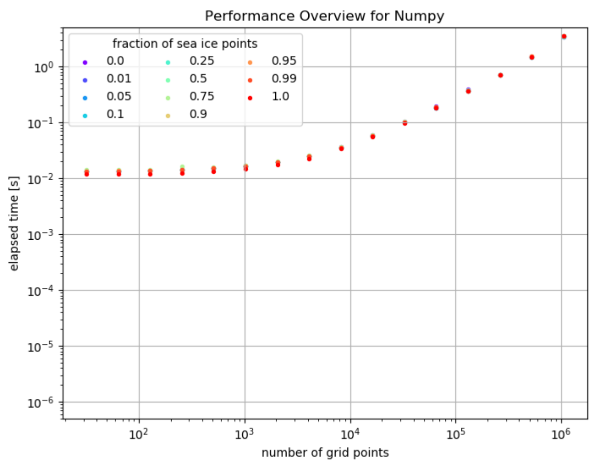
According to the results in Figure 1, for grid points up to 2^{14}, the sea ice parameterization runs fastest with Fortran. As no surprise, the optimized version of Fortran runs faster than the non-optimized version (see optimizing options in makefile on GitHub repository). Since the debugging decreases the performance of a code, the optimized Fortran version shows better performance results. Second in place is Python showing the best performance for numbers of grid points up to 2^{14}. Since Python is known to be in general slower than Fortran, this is to be expected. For larger grids (i.e. number of grid points $> 2^{14}$), the GT4Py backend *gtcuda* and Fortran (depending on the sea ice fraction) show the best performance. Also, the backend *gtx86* performs better for larger grids than Python.

However, it is surprising that for grid point numbers up to 2^{14}, all GT4Py backends show worse performances than Fortran and Python, even *gtcuda* running on GPUs. The bad performance of the GT4Py backends is due to the fact that the GT4Py DSL was written to optimize stencil operations over the entire three-dimensional grid for completely filled stencils. The sea ice parameterization can be very well represented in a stencil computation. However, sea ice is not present on a full 3D grid, but only in two dimensions. Moreover, for most sea ice fractions, many grid points were not covered with sea ice. Accordingly, operations are only performed on one part of the grid, which impacts the performance. It seems that for small grids, the performance cost of the GT4Py structure, i.e. storing the data in a stencil, is higher than the performance benefit of optimized stencil operation provided by GT4Py. There are also differences in the performance of the different GT4Py backends. As expected, *gtcuda* shows the best performance, since it runs on GPUs. It also seems reasonable that *gtx86* performs better than *numpy*, since it computes the stencil with C++ and not with Python. This explains why *gtx86* and *gtcuda* outperform Python for larger grids.

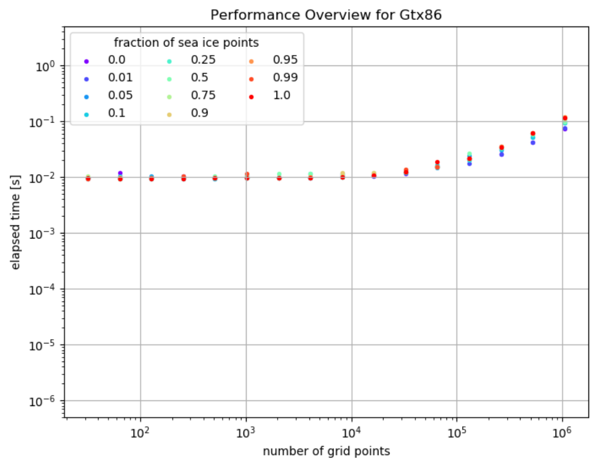
In Figure 1, we see that the elapsed time generally increases with the number of grid points used in the parameterization in all performance tests. For Fortran, the relation between the elapsed time and the number of grid points is approximately linear. For Python and GT4Py, however, this relation only becomes linear for larger grids. This indicates that the Fortran performance is mainly determined by the number of grid points, whereas grid points limit the performance of Python and the GT4Py backends only for larger grids. The dependence of the performance on the grid size implies that the performance is mainly determined by storing the grid and the calculations on the grid. As notable



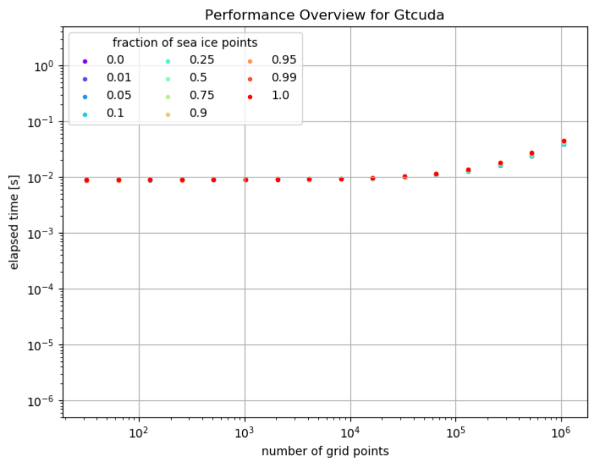
(a) Performance of plain Python



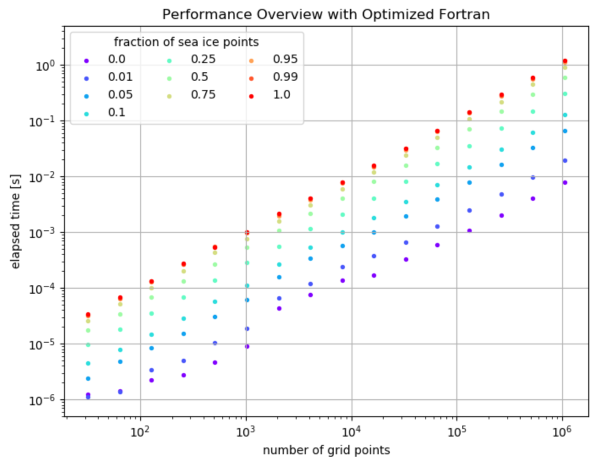
(b) Performance of GT4Py backend *numpy*



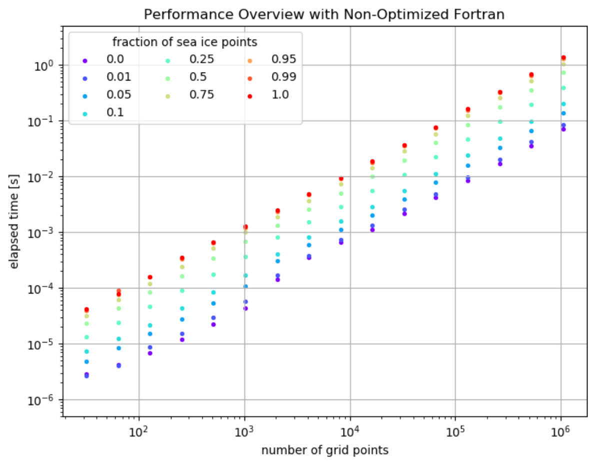
(c) Performance of GT4Py backend *gtx86*



(d) Performance of GT4Py backend *gtcuda*



(e) Performance of optimized Fortran

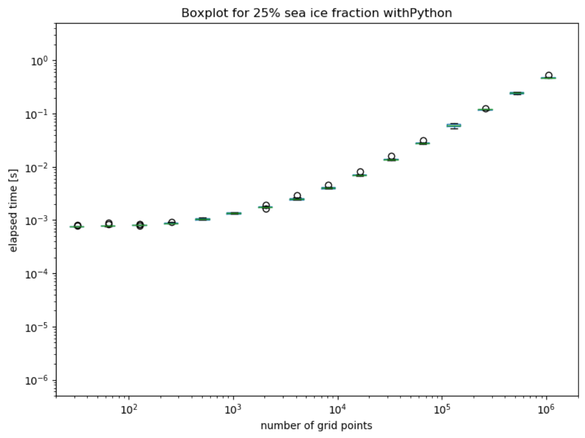


(f) Performance of non-optimized Fortran

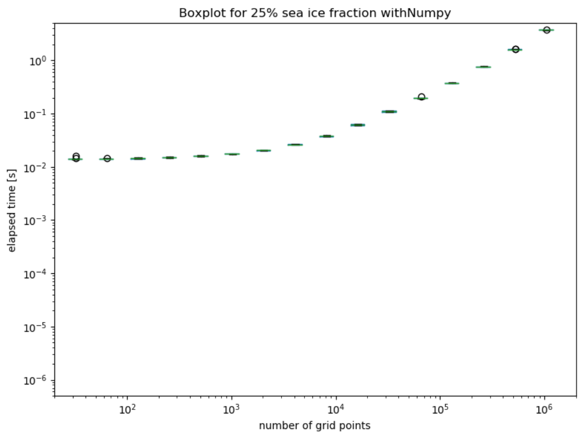
Figure 1: Performance of different GT4Py backends and Fortran as a function of number of grid points and fraction of sea ice points for the sea ice parameterization.

in Figure 1, the Python performance is limited by the grid size at a smaller number of grid points than the GT4Py backends are. This suggests that the GT4Py structure is useful for larger grids since the storing of stencils and the performing stencil operations do not cost much performance compared to the rest of the code. Comparing the different backends, we see that the performance of *numpy* is first limited by the grid size, whereas if the stencil is compiled with C++, the grid size becomes a performance-limiting factor only at 2^{14} and 2^{15} grid points for running on GPUs and CPUs, respectively. Since for the sea ice parameterization, calculations are only performed in grid points covered with sea ice, it can be expected that the performance decreases for an increasing sea ice fraction. For both Fortran compilations, this effect is visible. However, especially for small sea ice fractions, there are no big performance differences. The Python performance shows a weaker dependence on the sea ice fraction, demonstrating that grid point calculations cost relatively less in Python than in Fortran. The GT4Py backends show an even weaker dependence on the sea ice fraction. Interestingly, the backend *numpy* shows almost no variation in performance due to the sea ice fractions, although its performance is limited by the grid point number at smaller grid sizes than the other backends. This implies that the performance of *numpy* is rather limited by storing big stencils than by a huge number of stencil operations. This seems to apply to the other backends as well since their performance also varies more at larger grid points with the increasing grid size than with increasing sea ice fraction.

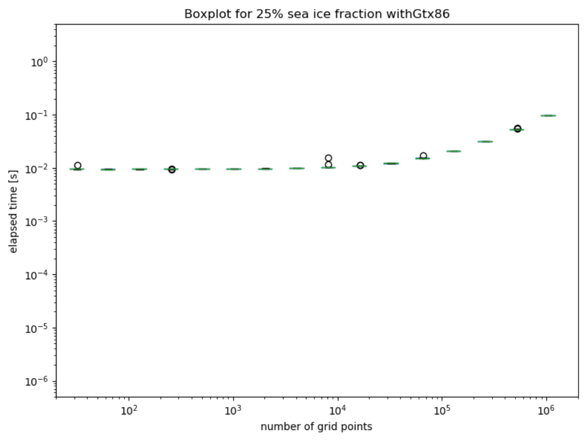
Figure 2 shows boxplots of the elapsed time according to grid size for 25 % grid point fraction with sea ice coverage. Overall, the elapsed time does not vary much for different iterations. For the GT4Py backends *numpy* and *gtx86*, there are almost no outliers. The boxplots of Python, the two Fortran compilations, and the GT4Py backends show up to 2 outliers. One of the outliers most probably corresponds to the first iteration. The first iteration has a longer elapsed time as it has to fetch the data to run the simulations, which are then stored in the cache and are much more readily available to conduct the computations. Therefore, several iterations are necessary to be computed to achieve a robust signal for the performances.



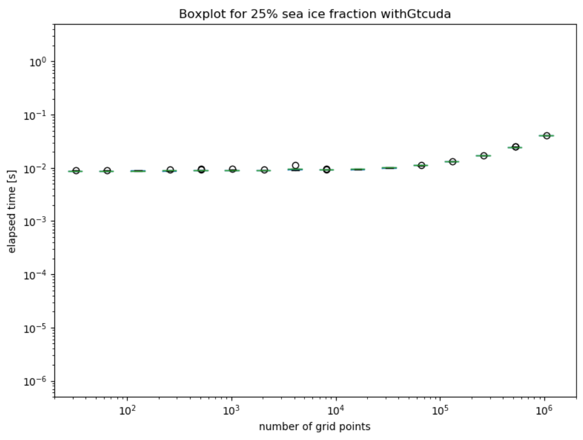
(a) Iterations with plain Python



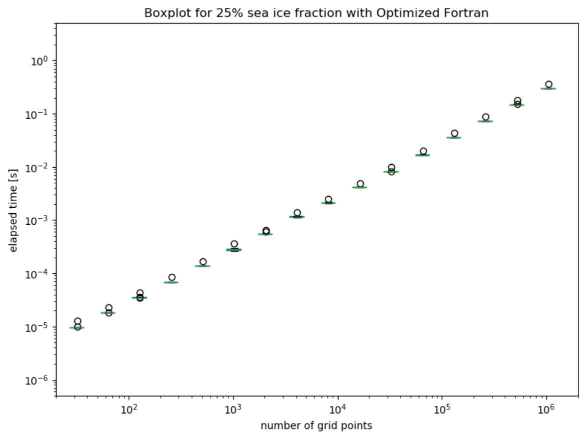
(b) Iterations with GT4Py backend *numpy*



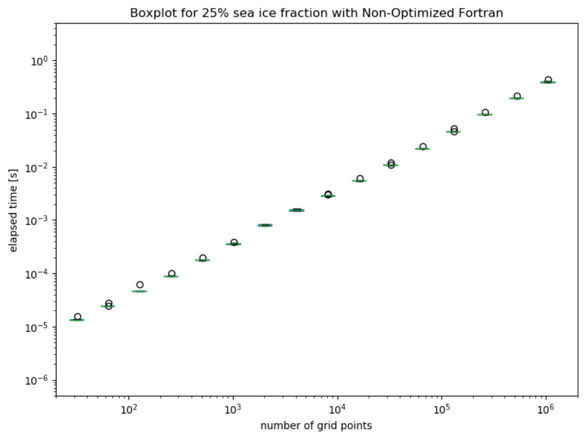
(c) Iterations with GT4Py backend *gtx86*



(d) Iterations with GT4Py backend *gtcuda*



(e) Iterations with optimized Fortran



(f) Iterations with non-optimized Fortran

Figure 2: Iterations with 25 % fraction of sea ice points with different GT4Py backends and Fortran.

3 Conclusion

This project contained the porting of a sea ice parameterization module from Fortran to Python and to Python's GT4Py library and compared the performances of different GT4Py backends. Two different Python implementations exist, one based on plain Python and one based on a GT4Py structure to test the performance on CPUs as well as GPUs. The results clearly underscore the popularity of Fortran for high-performance computing. The computation was up to the order of 10^3 faster than with Python and varying backends independently of being optimized or not. However, with increasing grid point numbers ($> 2^{14}$), the GT4Py backends *gtx86* and *gtcuda* show clear performance advantages. These two backends run both with C++ and *gtcuda* is executed on GPUs. Especially the performance of *gtcuda* highlights the potential of GPUs and GT4Py for climate modeling as larger grid point numbers are more efficiently computed.

References

Winton, M. (2000). A Reformulated Three-Layer Sea Ice Model. *Journal of Atmospheric and Oceanic Technology*, 17(4):525–531.