Writing a "modern" climate model using a DSL

Shallow convection parameterization

Langwen Huang, Chenwei Xiao, Mikael Stellio

Supervisor: Tobias Wicky

August 15, 2020

Contents

1 Introduction

NOAA's FV3GFS is a state-of-the-art global climate model developed in Fortran. In such complex atmospheric models, depending on the resolution of the computational grid used for the simulations, some relevant subgrid-scale processes are not resolved on the grid and require to be parameterized. Generally, processes to be parametrized include convection (shallow and deep), cloud cover, microphysical processes, boundary layer processes, radiation and other small scale processes that influence the larger scales and are relevant for a correct physical representation of the atmospheric system.
In this project, the GFS Scale-Aware Mass-Flux (SAMF) shallow convection parameterization scheme will be ported from its original Fortran code to Python using GT4Py for stencil computations, and the performance for different backends will be explored.
GT4Py (GridTools For Python) is a domain-specific library that exposes the Python embedded domain-specific language (DSL) GTScript tailored for high-performance implementation of stencil kernels from a high-level definition. It is important to note that GT4Py allows for compilation with different backends to generate optimized code for a specific target architecture. Available Python backends are *"debug"* (for loops) and *"numpy"* (vectorized syntax), while C++ backends are *"gtx86"* (x86), *"gtmc"* (MIC), *"gtcuda"* (CUDA-enabled Nvidia GPUs)[3] . Finally, GT4Py also provides storages with optimal memory strides, alignment and padding to increase memory accesses performance within stencil kernels.

1.1 Parameterization scheme

The GFS SAMF shallow convection scheme has scale and aerosol awareness, and parameterizes the effect of shallow convection on the environment. It is similar to the GFS SAMF deep convection scheme, but it only has the "static" and "feedback" control parts.
The shallow convection algorithm is a simplified form of the deep convection scheme, where no quasi-equilibrium assumption is made for small grid sizes[1] , no convective downdrafts are taken into account, the entraiment rate is greater and shallow convection is confined below a pressure level that corresponds to 70% of the surface pressure.
Furthermore, the parameterization scheme was divided in the following four steps[2] :

- Initialization: *"Compute preliminary quantities needed for the static and feedback control portions of the algorithm."*
- Static control: *"Perform calculations related to the updraft of the entraining/detraining cloud model."*

- Compute tendencies: *"Calculate the tendencies of the state variables (per unit cloud base mass flux) and the cloud base mass flux."*
- Feedback control: *"Calculate updated values of the state variables by multiplying the cloud base mass flux and the tendencies calculated per unit cloud base mass flux from the static control."*

2 Structure of the code

During the course of the project, first a working serialization to read input and reference output data was implemented and secondly different sections of the Fortran code were assigned to be ported, while a dependency analysis was performed. GT4Py stencils were implemented to execute the basic operations in the form of forward, backward or parallel computation blocks in spite of normal Python for-loops. The stencil implementations can be found in folder */shalconv/kernels*, where also a file containing some utility GT4Py functions (min, max, log, ...) is present, while the dependency analysis code and *.xml* files can be found respectively in */tests/analyse_xml.py* and the */tests/fortran* folder.
Moreover, a GT4Py function `fpvsx_gt()` to compute exact values of saturation vapor pressure from temperature was implemented, since for simplicity it has been decided to not follow the lookup-table approach adopted in the original Fortran code. The function implementation can be found in file *funcphys.py*.
The project followed a test-based approach in order to verify each of the newly ported four parts of the scheme listed in section 1.1 independently and facilitate the localization and resolution of errors.
The stencils tests were based on a comparison between the evolving fields obtained from the reference Fortran code and the results of the GT4Py implementation. Four serialization breakpoints were set at the end of each of the four parts and after the execution of the implemented kernels the fields were validated against the read references. All tests can be found in the */tests* folder.
In conclusion, main files to check for validation (*main.py*) and produce benchmark results (*benchmark.py*, *plot.py*) can be found in the main folder.
As a final remark, it is important to point out that GT4Py kernels have also been written for subroutine `samfaerosols()`, but, due to it never being called during the specific program run to validate and the lack of reference results for it, these kernels were never checked for correctness and might not be up to date with the rest of the code.

3 Coding challenges

Currently GT4Py is still in development, therefore it is no surprise to encounter some technical problems or inconveniences, that will hopefully be improved or supported by the domain-specific language in the future.
In this section the main coding challenges are explored and possible solutions are discussed.

3.1 Shape of the fields

As for now GT4Py only supports fields of the same three-dimensional shape to be used in a specific stencil. However, the original Fortran code contains a lot of interactions between both one-, two- and three-dimensional arrays. This implies an unconvenient redesign of all the different fields to a common shape `(1, ix, km)`, thus one-dimensional arrays were duplicated along the k-dimension (see section 3.2) and three-dimensional array, generally storing values for different independent tracers, had to be treated in slices of the correct shape (see section 3.3).
Moreover, the function `numpy_dict_to_gt4py_dict()` was implemented to transform a numpy array into GT4Py storages.

3.2 Coherence of 1D arrays

As aforementioned, all one-dimensional arrays are transformed into three-dimensional fields of shape `(1, ix, km)`, where each k-level should represent the same original one-dimensional array. To ensure that stencil computations involving both 1D and multi-dimensional arrays always access a memory address holding the correct value for the current index, special care must be taken for dependecies in 1D arrays assignments.

```
1  do k = km, 1, -1
2    do i = 1, im
3
4      if (cnvflg(i)) then
5        if(k < ktcon(i) .and. k > kb(i)) then
6
7          rntot(i) = rntot(i) + pwo(i,k) * xmb(i) * 0.001 * dt2
8
9        endif
10     endif
11
12   enddo
13 enddo
```

The code above shows a situation where a one-dimensional array is updated using values from its previous state and values from a two-dimensional array. It is important to notice that the 1D array `rntot` is updated in a nested loop along both dimensions i and k, thus a specific location i could be potentially

be accessed and modified `km` times.
On the other hand, in GT4Py stencils 1D arrays are represented by 3D fields, thus to ensure equality of all k-levels the following solution is proposed, since the expression at line 7 evaluates differently for each vertical level.

```
1  with computation(BACKWARD):
2
3      with interval(-1, None):
4
5          if cnvflg == 1:
6              if k_idx > kb and k_idx < ktcon:
7                  rntot = rntot + pwo * xmb * 0.001 * dt2
8
9      with interval(0, -1):
10
11         if cnvflg == 1:
12             if k_idx > kb and k_idx < ktcon:
13                 rntot = rntot[0, 0, 1] + pwo*xmb*0.001*dt2
14             else:
15                 rntot = rntot[0, 0, 1]
16
17 with computation(FORWARD), interval(1, None):
18
19     if cnvflg == 1:
20         rntot = rntot[0, 0, -1]
```

A forward/backward propagation approach is used. In a first step the computations are performed backward along the k-levels (lines 1-15) ensuring a correct access order of previous `rntot` values, while in a second step the final result on the last k-level is propagated forward (lines 17-20) to guarantee the coherence of the 3D field along all vertical levels. As a final remark, notice that due to dependencies these situations limit the usage of the parallel computation block and forces to interrupt or split any non-backward or non-forward computation block present in the kernel.

3.3 Tracer loops

As mentioned in section 3.1, three-dimensional arrays of shape `(ix, km, ntr)` or `(ix, km, ntc)` containing values of a specific variable for different independent tracers have to be treated in slices of shape `(1, ix, km)` to be consistent with the previously defined representation of fields in GT4Py stencils.

```
1 for n in range(ntr):
2     ctro_slice[...] = ctro[np.newaxis, :, :, n]
3
4     stencil_ntrstatic0(cnvflg, k_idx, kmax, ctro_slice)
5
6     ctro[:, :, n] = ctro_slice[0, :, :]
```

The example above shows a typical treatment of a loop over tracers. It is important to point out that for each tracer the corresponding slice of the

full 3D numpy array is copied into a GT4Py storage and after the stencil computations the results are copied back in the right location. Due to dependecies in the data used by the kernels (e.g. `cnvflg`) this procedure has to be repeated for each tracer loop, implying frequent memory copies between numpy arrays and GT4Py.

3.4 Data dependency

To guarantee the correct ordering, arguments and inputs of stencils, explore the possibility of merging all tracer loops into a single loop at the end to avoid copying data and have a better understanding of the whole code, a dependency analysis was conducted.
In a first step the open-fortran-parser was used to generate an abstract syntax tree (AST) of the original Fortran subroutine `samfshalcnv()` in XML format. Secondly, running *analyse_xml.py* a scan of the *.xml* file was performed to detect assignments and reads of each variable in a given range. In this way the input and output variables to be serialized to validate each of the four parts of the code (see section 1.1) were found.
Moreover, the approach explained above is also useful to find the dependency tree of a variable at a given line, making it easier to trace errors, and by traversing the XML tree stencils for each loop could be generated automatically.
In conclusion, this revealed the impossibility of merging tracer loops, provided useful information to reorder some computations, revealed the correct input and output arguments required by each portion of the code to be tested and gave insights on where a forward or backward computation block should be used in spite of a parallel one.

3.5 Temporary variables

Inconveniently, GT4Py does not support the definition of temporary variables inside if-statements, as shown in the example below.

```
with computation(BACKWARD), interval(0,-1):

    if cnvflg == 1 and k_idx < kbcon and k_idx >= kb:

        dz   = zi[0, 0, 1] - zi
        ptem = 0.5 * ( xlamue + xlamue[0, 0, 1] ) - xlamud

        eta  = eta[0, 0, 1]/( 1. + ptem * dz )

with computation(PARALLEL), interval(0,-1):

    if cnvflg == 1 and k_idx <= kmax - 1:

        qeso = 0.01 * fpvs(to)
```

In this case, lines 5 and 6 will raise an error due to the definition of temporary variables `dz` and `ptem`. This could be solved by moving the initialization of the two variables outside of the conditional statement and only in a second time assign the correct values to them.

```
1  with computation(BACKWARD), interval(0,-1):
2
3      dz   = 0.0
4      ptem = 0.0
5
6      if cnvflg == 1 and k_idx < kbcon and k_idx >= kb:
7
8          dz   = zi[0, 0, 1] - zi
9          ptem = 0.5 * ( xlamue + xlamue[0, 0, 1] ) - xlamud
10
11         eta  = eta[0, 0, 1]/( 1. + ptem * dz )
```

Moreover, more complex situations involve the use of GT4Py functions that require local variables inside their scope, thus causing errors when called in if-statements, as for `fpvs()` at line 14. Similarly as before, this could be solved by calling the function outside the conditional statement and assigning its return value to a temporary variable, although, depending on the computational cost of the function, this would imply unnecessary costly computations for some grid-points in the case where the condition on line 12 is not satisfied.

```
1  with computation(PARALLEL), interval(0,-1):
2
3      tmp = fpvs(to)
4
5      if cnvflg == 1 and k_idx <= kmax - 1:
6
7          qeso = 0.01 * tmp
```

4 Results

Firstly, it is important to point out that all fields resulting from the GT4Py implementation validate with the reference Fortran results.
In this section benchmarks for different GT4Py backends (*"numpy"*, *"gtx86"* and *"gtcuda"*) are compared to the performance of the original Fortran code for different grid sizes on a Piz Daint node with the following specifics.

- CPU: Intel(R) Xeon(R) CPU E5-2690 v3 @ 2.60GHz
- Thread(s) per core: 2
- Core(s) per socket: 12
- Total memory: 65.842 GB
- GPU: Tesla P100-PCIE (Driver version: 418.39, CUDA version: 10.1)

Note that only the number of columns was modified, while the vertical levels remained constant.
For each benchmark N columns were randomly selected among all the columns available from the serialized data provided, divided in 6 tiles with 19 savepoints containing 2304 columns each, thus a total of $6 \cdot 19 \cdot 2304 = 262656$ columns. Subsequently, everything was fed to the GT4Py and Fortran programs and the elapsed time was obtained for both. This procedure was repeated 10 times for each benchmark and the mean timings and corresponding 95% confidence interval, computed based on the student's t-distribution

$$\mu_t \pm A \cdot \frac{\sigma_t}{\sqrt{10}}, \quad \text{where } A = 2.262,$$

were plotted, as shown in figure 1.

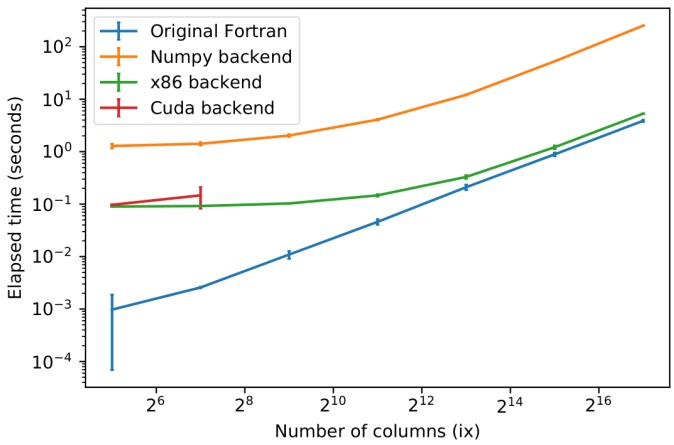

Figure 1: Comparison between different GT4Py backends and Fortran

It can be observed quite easily that the Fortran code still dominates performance-wise, being approximately 100 times faster than the *"numpy"* backend. However, it is important to notice that the performance of the C++ backend *"gtx86"* is comparable to the original code for a large number of columns, supporting the idea that porting atmospheric models to Python with the help of domain-specifc languages like GTScript is possible, without losing too much performance.
In general, with increasing `ix` the runtime for Fortran increases linearly, but this is not always the case for GT4Py backends. Figure 1 shows that for small numbers of columns the GT4Py implementation reaches a lower boundary in execution time, that could be explained by the overhead for calling a GT4Py stencil (note that stencils also have a large number of field arguments, that could have an impact on performance) being larger than the actual computation time. In fact, for larger numbers of columns the GT4Py backends timings follow the same linear behavior observed in the Fortran benchmarks, meaning that the computation time dominates for larger grids. As a final remark, it is important to point out that for the *gtcuda* backend only timings for $ix <= 128$ are visible due to synchronization and "out of memory" issues that could not be solved (in time).

5 Future work

The current state of the code is still open to some optimizations and improvements to increase the efficiency of the GT4Py stencils.
Further investigation on the forward/backward propagation approach used could reveal a more efficient method and more thorough dependency analysis could encover the possibility to merge some forward and backward blocks.
The number of operations performed for computations involving only one-dimensional arrays, corresponding to non-nested 1D loops in the Fortran code, could be drastically reduced by calculating all the relevant values only on the first k-level and subsequently propagating the results to all the other vertical levels to ensure the coherence of the GT4Py field (see section 3.2), instead of re-evaluating the same expressions for multiple k-levels.
In summary, even though results comparable to the original Fortran timings were obtained through the *"gtx86"* backend, there is still room for optimization on the GT4Py code.

6 Conclusions

In conclusion, the results of this project show that porting atmospheric models, generally written in Fortran, to Python without drastically impacting the performance of the code is a possibility, and with the help of the different backends offered by GT4Py faster and more efficient code could also probably be obtained, e.g. *"gtcuda"*, especially if kernels are written intelligently and efficiently (see section 5 for some examples). Moreover, since GT4Py is currently in development it could prove itself even more advantageous for Python users in the future, in terms of both performance gains and user accessability (see section 3).

References

[1] Han, J., Wang, W., Kwon, Y. C., Hong, S.-Y., Tallapragada, V., Yang, F., 2017: Updates in the NCEP GFS Cumulus Convective Schemes with Scale and Aerosol Awareness. Wea. Forecasting, **32**, 2005-2017.

[2] CCPP Scientific Documentation

[3] ETHZ High Performance Computing for Weather and Climate lecture notes