GT4Py vs. Fortran

High Performance Computing for Weather and Climate Models

Charles Pierce
Lena Brun
Lena Fasnacht
Floriane Gabriel

July 2024

1 Introduction

Understanding and predicting weather and climate patterns on Earth is crucial for various industries, such as agriculture and energy production, and plays a vital role in public safety. Accurate weather forecasts and climate predictions can inform agricultural practices, optimize energy resource management, and alert the public to extreme weather events, potentially saving lives and reducing economic losses [7].
The governing equations of the climate system pose a complex partial differential equation (PDE) problem, which cannot be solved analytically. In order to solve the PDEs numerically in a model, stencil computation algorithms are used. These algorithms are both space and time discrete, meaning that the model variable values at each grid point are computed based on the neighboring grid point values [2].
In order to get accurate results a fine grid resolution is key. Reducing the grid size in forecasting models has significantly improved their accuracy. However, halving the grid spacing leads to an eightfold increase in required computational power [7]. High performance computing looks to aggregate computing power in a way that improves performance of the available (super)computer architecture[2] and in so doing run high resolution climate and weather models in the most efficient way possible.
This project aims to investigate the impact different processing units, coding languages, and backends have on stencil computation run times by doing an in-depth performance analysis for a 4^{th} order diffusion stencil program.

2 Methods

2.1 Languages

2.1.1 GridTools for Python

GridTools for Python (GT4Py) is a domain specific language to deal with high performance numerical stencil computations on grid-based problems. It allows using the C++ GridTools ecosystem library that provides efficient tools for high performance stencil computations and other grid based operations in a Python interface, i.e., with Python syntax and semantics. This increases readability and simplifies the handling, while benefiting from the performance optimizations available in GridTools [2].

2.1.2 Fortran

Fortran is the most widely used high-level, scientific programming language specifically designed for numerical and scientific computations. It is known for its efficient handling of array operations and mathematical computations,

as it allows for performance optimization through vectorization and parallelism. Fortran is compatible with both CPUs and GPUs and thus enables high performance computing across different hardware platforms. Fortran's popularity mainly originates in its high readability while making significant performance improvements in scientific and engineering applications possible [1].

2.2 Processing Units

2.2.1 Central Processing Unit (CPU)

The CPU is a versatile and powerful component designed to handle a wide range of general-purpose tasks. In weather and climate modeling, CPUs are responsible for data processing and execute the numerical calculations necessary to solve the PDE problem. CPUs ensure the reliability and accuracy needed in weather and climate prediction, making them the default processors for weather and climate models [4].

2.2.2 Graphics Processing Unit (GPU)

The GPU is a specialized processor designed to accelerate graphics rendering and parallel processing tasks. GPUs are increasingly integrated into weather and climate modeling, because they are particularly good at parallel data processing due to their architecture, which consists of many smaller, efficient cores. Due to this architecture GPUs can significantly decrease the model run times by solving large matrix equations, processing atmospheric dynamics, or handling radiative transfer calculations in parallel. So while CPUs are the default for existing weather and climate models, GPUs have great potential for future models with smaller spatial and temporal resolutions [4].

2.3 Experiments

To achieve an optimal performance for our diffusion stencil computation, we conduct a formal analysis of the effect of different backends and domain sizes on the processing time of our diffusion stencil program. The setup of the different program versions will be explained in the following sections.

2.3.1 Fortran Backends

Two program versions for the execution on CPU were developed, one is using OpenMP, while the other one is using CoArray and both are combined with MPI. Fortran CoArrays allow parallelization through the sharing of variables between different images, which can be considered as independent threads of execution [6]. OpenMP is an API which allows shared-memory

multiprocessing and is thus an easy way to parallelize parts of the stencil program [2]. MPI is an abbreviation for Message Passing Interface which allows data to move from one processes's adress space to another using cooperative operations [3].

The Fortran code was then adapted to work in parallel on the GPU using OpenACC. OpenAcc works in a similar fashion as OpenMP, in that it uses simple instruction lines in the code to indicate where the parallelisation should take place [2]. One great advantage of using OpenAcc is the memory management between the CPU and the GPU. It uses specific keywords to handle the allocations and deallocations of memory as well as the data transfer in both directions [5]. Two versions of this code were developed with varying degrees of parallelisation. The first version follows the k-parallelisation used on the CPU with OpenMP. The second version goes further and pushes the parallelisation into the i and j loops as well as the function in charge of updating the halo.

2.3.2 Gt4Py Backends

Three GT4Py backends were chosen to compare performance with Fortran on both GPU and CPU. The CPU backends `"gt:cpu_ifirst"` and `"gt:cpu_kfirst"` were chosen. They both compile C++ GT-based code for CPUs [2], however k-first is designed specifically to run on multiple cores. When executing code, they differ in that `"gt:cpu_ifirst"` runs through variables in a `i-k-j` order and `"gt:cpu_kfirst"` does so in a `k-j-i` order. The `"gt:gpu"` backend was chosen so as to get a comparable C++ GT-based code for GPUs. Superior performance is expected from all GT4Py backends compared to Fortran code, seeing as this domain-specific Python library was created to do just that [8].

2.3.3 Domain Size Configurations

In our experiments we compare six versions of our diffusion stencil: OpenMP, CoArray, GT4Py k-first, GT4Py i-first, GT4Py on GPU, and OpenAcc.
For the domain size configurations, we started by fixing $n_y = 64$ and $n_z = 32$ while varying $n_x = 2^4, 2^6, 2^8, 2^{10}, 2^{12}, 2^{14}, 2^{16}$ in Figure 1. This gave us $2^{15}, 2^{17}, 2^{19}, 2^{21}, 2^{23}$, and 2^{25} grid points. By keeping two dimensions constant and increasing n_x we can quantify the impact the total number of grid points has on the run time per grid points for each of the different versions, while having a simple experimental setup. It allows us to control that there are no additional sources of variation impacting our result. By doing the analysis for each of the directions and receiving similar but not identical results every time, we can distinguish the impact the number of grid points and the dimensionality of the grid points have on the performance of the versions. We then repeated the experiment in Figure 2 for $n_x = 64$ and

$n_z = 32$ while varying $n_y = 2^4, 2^6, 2^8, 2^{10}, 2^{12}, 2^{14}, 2^{16}$, which again gave us $2^{15}, 2^{17}, 2^{19}, 2^{21}, 2^{23}$, and 2^{25} grid points. And lastly, $n_x = 64$ and $n_y = 64$ while varying $n_z = 2^2, 2^4, 2^6, 2^8, 2^{10}$, which again gave us $2^{13}, 2^{15}, 2^{17}, 2^{19}$, and 2^{21} grid points in Figure 3.

3 Results and Discussion

3.1 Runtime per Grid Point

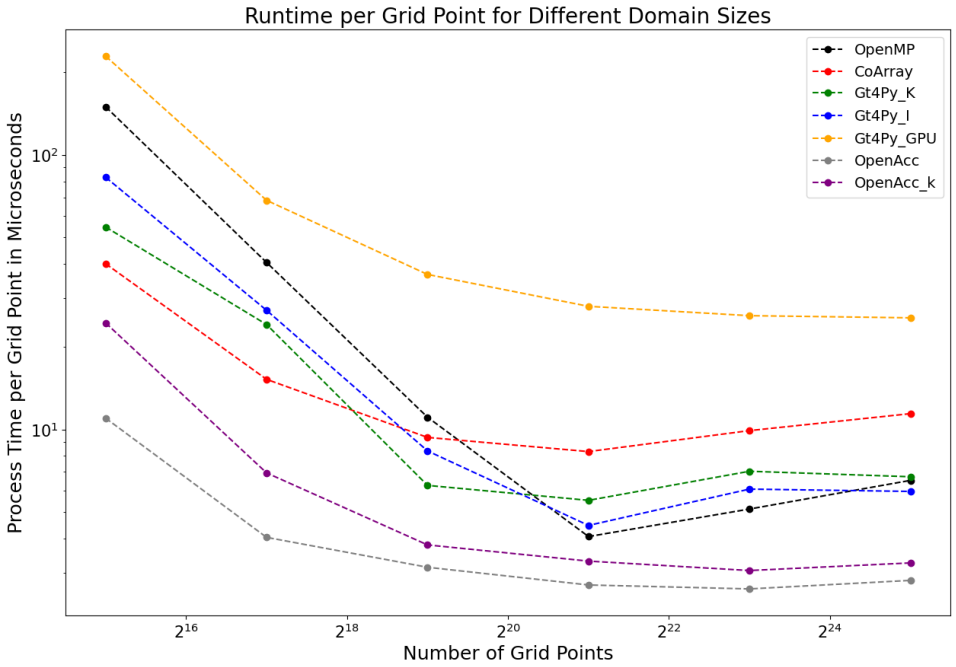

Figure 1: Runtime per grid point in microseconds for different total numbers of grid points with varying nx values while keeping $ny = 64$ and nz $= 32$ in a loglog plot

Figure 1 shows the runtime per grid point for different domain sizes for each of our stencil versions. The domain size is altered by giving nx different values, while ny and nz remain the same. We observe a general trend of an initially decreasing runtime per grid point. However, for more than 2^{21} grid points this trend seems to be reversed and the runtime per grid point increases again or stabilizes. There are a few possible reasons for this behaviour. First of all, more grid points mean that running the stencil program is more computationally expensive and we would therefore intuitively expect a monotonically increasing curve. However, the methods we are using are specifically made for large computational tasks, which is why computational

resources are used more efficiently as the grid point number increases. If we for example look at the parallelization done in the OpenMP version of the code, communication and synchronization processes become less significant relative to the total computation time for larger domain sizes.
The observed increase/stabilization of runtime per grid point can be explained by cache and memory hierarchy. As the grid size is relatively small, much of the data can fit into the faster levels of the memory hierarchy (e.g., L1,L2, and L3 caches on the CPU). This way memory access is done efficiently and there is a lower process time per grid point. Once the domain size becomes too large for the cache size, more data needs to be imported from the main memory. This increases memory access time and can cause the runtime per grid point to stabilize or even increase. The system is thus limited by memory bandwidth rather than computational power.
Runtime differences between the different stencil versions vary up to factor 10. The overall fastest version for every number of grid points is the OpenAcc fully parallelised version, while the GT4Py GPU version has the longest runtime per grid point for every number of grid points we considered. These two versions are interestingly also the ones exhibiting a stabilization for the largest numbers of grid points considered in our experiments. All other versions show a very steep decrease until the aforementioned 2^{21} grid points, beyond which the process time per grid point starts to increase again. Most of the versions maintain their pace relative to the others, except for CoArray, which performs relatively well for smaller domain sizes and then ends up being the second slowest for the largest number of grid points we considered. The OpenMP version shows an impressive increase in performance once it deals with larger domains but shows signs of a steeper increase when domain continues growing compared to the milder increase or stabilisation the other versions show. Interestingly, the size of the domain also makes a difference in the comparison of the two CPU GT4Py versions. The GT4Py k-first program performs better initially, but the GT4Py i-first program shows a stronger improvement of runtime per grid point and from 2^{20} is the faster of the two, which makes sense given that in this case we increased the number of grid points in the x direction, and as such the memory is being accessed more efficiently by iterating in the x direction first in the case of the `gt:cpu_ifirst` backend.

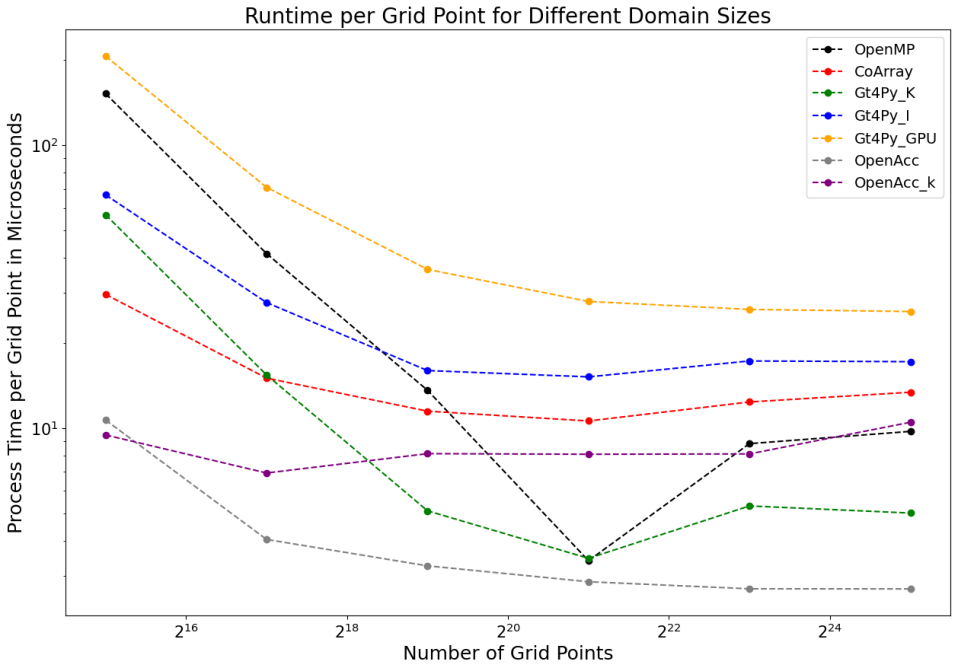

Figure 2: Runtime per grid point in microseconds for different total numbers of grid points with varying ny values while keeping $nx = 64$ and nz = 32 in a loglog plot

Figure 2 shows the different runtime per grid point but this time, the domain size is altered by varying the variable ny instead of nx. Even though the same numbers of grid points are analysed, the run times are very different from the ones in Figure 1. The best and worst performer remain the same but the change in performance of the other versions is very different. While the OpenMP and GT4Py k-first versions retain their steep increase in performance per grid point for an increasing domain, the GT4Py i-first as well as the OpenACC k-parallel and the CoArray versions show an earlier stabilisation. Interestingly, the OpenMP outperforms the OpenACC-k at 2^{21} to then match the runtime per grid point for larger domains. This is surprising as the two programs are designed with the same parallelisation, which indicates that this parallelisation does not allow a decrease of the overhead for larger domains in the GPU program. The GT4Py implementations show a strong difference between looping first in k or in i as the i-first implementation shows only a weak decrease in runtime per grid point. This is understandable as i-first version is designed in such a way the accessing the memory becomes more efficient for an increase of grid point in the i direction, namely nx. Indeed, as grid points are being exponentially increased in the y direction and this backend iterates in the y direction last, this loss in efficiency compared to the k-first backend is sensible.

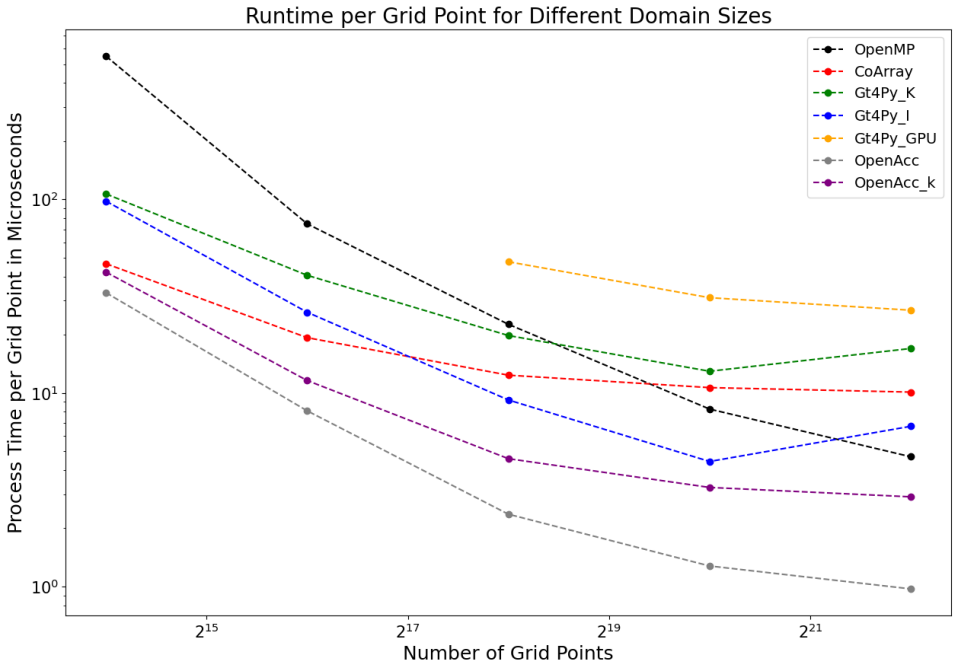

Figure 3: Runtime per grid point in microseconds for different total numbers of grid points with varying nz values while keeping $nx = 64$ and ny $= 64$ in a loglog plot

In Figure 3 we now see the runtimes per grid point for varying nz values. We again observe some similar traits to the previous two Figures: Best performing is the OpenAcc fully parallelised version of the stencil, while GT4Py ran on the GPU has the highest runtime per grid point for all considered numbers of grid points.
We didn't consider as many levels for nz as we did for the other two dimensions, since typically less vertical levels are considered in weather and climate models due to the isotropy of the atmosphere. As a result, there are smaller numbers of total grid points. We can therefore not see the previously observed increase in runtime per grid point due to memory bandwith limitations. We merely see a stabilization and a slight increase for the GT4Py i-first version.

4 Conclusion

This comparison study highlights a few interesting aspects that can come into play when choosing how to implement a stencil computation. First, the size of the domain matters. When restricted to a CPU, a CoArray solution on Fortran yields a better performance for smaller domain sizes than an Fortran-OpenMP solution or an implementation on GT4Py. However each

of these can perform better for larger domain provided they are structured appropriatly for the domain in question. This leads to the second point, namely building the right implementation for the given domain. There is a significant difference between the i-first or k-first versions of GT4Py depending on whether the domain extends more the x or the y direction, so this too needs to be taken into consideration. While the OpenACC solution shows a great overall performance, it is dependent on Fortran or C++, where the GT4Py solutions allow for a more user friendly implementation on python, more commonly used and understood. The best way to get an ideal performance would of course still be to tailor custom the solution to the architecture of the machine on which the computation will be done. Understanding especially where the bottlenecks are, in terms of memory and transfer between the CPU and GPU, is crucial in designing a best performance implementation.

References

[1] J. W. Backus and W. P. Heising. Fortran. *IEEE Transactions on Electronic Computers*, EC-13(4):382–385, 1964.

[2] Oliver Fuhrer. Lecture slides. Lecture notes, 2024. High Performance Computing for Weather and Climate Models, ETH Zürich.

[3] Lawrence Livermore National Laboratory. What is mpi? https://hpc-tutorials.llnl.gov/mpi/what$_i s_m pi/$, 2024. *Accessed* : $2024 - 08 - 25$.

[4] Zulfiqar A. Memon, Fahad Samad, Zafar Rehman Awan, Abdul Aziz, and Shafaq Siraj Siddiqi. Cpu-gpu processing. *INTERNATIONAL JOURNAL OF COMPUTER SCIENCE AND NETWORK SECURITY*, 17(9):188–193, SEP 30 2017.

[5] M. Norman, J. Larkin, A. Vose, and K. Evans. A case study of cuda fortran and openacc for an atmospheric climate kernel. *Journal of Computational Science*, 9:1–6, 2015.

[6] Paul Norvig. Parallel programming with coarrays in fortran, 2024. Accessed: 2024-08-23.

[7] Christoph Schär, David Leutwyler, and Martin Wild. Lecture slides. Lecture notes, 2024. Weather and Climate Models, ETH Zürich.

[8] S. Ubbiali, C. Kühnlein, C. Schär, L. Schlemmer, T. C. Schulthess, M. Staneker, and H. Wernli. Exploring a high-level programming model for the nwp domain using ecmwf microphysics schemes. *Geoscientific Model Development Discussions*, 2024:1–30, 2024.