Performance Comparison of High-Level Programming Languages in Python

Tom Lausberg, Tobia Claglüna, Safira Piasko

August 2021

Abstract

Various scientific fields have benefited greatly from easy-to-use domain specific languages (DSL). GT4Py aims offer a similar high-performance, high-level solution for weather and climate simulations. It implements an extension of Python which generates efficient implementations of finite difference stencil computations for a selection of backends and architectures. In this report we will explore the performance of GT4Py and how it compares to a Machine Learning DSL and several general purpose optimized NumPy implementations. We find that GT4Py is superior to the other options in a variety of situations and offers a 4x speedup on CPUs and 2x speedup on GPUs when compared to the next best library that we test.

Chapter 1

Introduction

Weather and climate simulations suffer from an insatiable demand for more computing power, higher resolutions, and more expensive parameterizations. This means that modern simulation codes need to be highly optimized to achieve acceptable levels of runtime. To create such a code, a multi-step workflow is usually applied. After initially implementing a prototype in a high-level language, the code is rewritten in a faster, more complex low-level language. This code can then be optimized for optimal cache usage and single core performance. Furthermore, it can be modified to allow for multi-thread parallelization, NUMA awareness, GPU acceleration and heterogeneous computing. Each of these steps are time consuming and allow for the introduction of bugs. The development cycle also severely limits the ability to modify a code and requires specific implementations for most hardware.

Domain specific languages aim to offer a solution to the aforementioned challenges. By providing a programming language tailored and limited to a particular application domain, domain specific languages pursue the goal of creating a fast, maintainable and hardware agnostic language. DSLs allow domain experts to quickly write high-level declarative code and then in a single step target a specific backend or environment. Whilst DSLs offer major advantages, they themselves require major efforts to be developed and maintained. In other fields such as machine learning, advanced libraries and DSLs have been very successful. Large companies and research organizations have invested immense amounts of resources to create highly optimized and efficient packages. In this project we will compare a baseline NumPy implementation of two computational stencils with implementations in a weather and climate DSL, general HPC NumPy implementations and a machine learning DSL designed for automatic differentiation.

Chapter 2

Implementation

2.1 Operators

Weather and climate simulations require numerically solving partial differential equations (PDEs). The application of differential operators to grid point values can become the bottleneck of the models. GT4Py's main task is to optimize the operators by using stencil calculations. To test the performance of GT4Py, we compared four differential operators each implemented in the four different languages mentioned before.

One of the most used differential operators is the second order Laplace operator. In two dimensions the Laplace operator

$$\Delta = \nabla^2 \overset{\text{2D}}{=} \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} \tag{2.1}$$

applied on a variable f can be discretized to

$$(\Delta f)_{i,j} \approx \frac{1}{(\Delta x)^2}(-4f_{i,j} + f_{i-1,j} + f_{i+1,j} + f_{i,j-1} + f_{i,j+1}). \tag{2.2}$$

As most governing equations in weather and climate models are three dimensional, we also consider the three dimensional Laplace operator

$$\Delta \overset{\text{3D}}{=} \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2} \tag{2.3}$$

The stencil operation on grid point $x_{i,j,k}$ is defined by

$$\begin{aligned}(\Delta f)_{i,j,k} \approx \frac{1}{(\Delta x)^2}(-6f_{i,j,k} + f_{i-1,j,k} + f_{i+1,j,k} \\ + f_{i,j-1,k} + f_{i,j+1,k} \\ + f_{i,j,k-1} + f_{i,j,k+1}).\end{aligned} \tag{2.4}$$

Additionally we analyse a computationally more expensive fourth order differential operator called the Biharmonic operator.

$$\nabla^4 = \Delta^2 \tag{2.5}$$

A naive implementation of the Biharmonic operator consists of the Laplace operator being applied twice. Hence by comparing the runtime of the Laplace and the Biharmonic operator we can observe the optimizations each of the individual libraries are able to apply.

2.2 Libraries

GT4Py

GT4Py [7] is a domain-specific library currently under development and exposed by the domain-specific language GTScript which is embedded in Python. The goal of this DSL is to provide a user-friendly, declarative programming language specifically designed for weather and climate applications. A key feature is performance portability, with various backend compilation options included to generate and optimize code for different architectures: `"NumPy"` generates a Python code through vectorized syntax, `"gtx86"` produces C++ code with OpenMP parallelism for ij-blocking and `"gtcuda"` creates CUDA code to be run on GPUs [9].

Weather and climate models are generally based on stencil computations. Therefore, GT4Py focuses on generating stencil kernels. These kernels are defined by decorating a function, depicted in Listing 2.1.

```
@gtscript.stencil(backend="numpy")
def laplacian(in_field: Field[np.float64],
              out_field: Field[np.float64]):
    with computation(PARALLEL), interval(...):
        out_field = (-4. * in_field[ 0,  0, 0]
                     +     in_field[-1,  0, 0]
                     +     in_field[+1,  0, 0]
                     +     in_field[ 0, -1, 0]
                     +     in_field[ 0, +1, 0] )
```

Listing 2.1: Example of a Laplace stencil

The implementation of the operators presented in Chapter 2.1 can be accomplished very efficiently. Since GT4Py focuses precisely on these kinds of stencils, many user-friendly features are built in. For example, handling the domain size to which the stencil can be applied can in general be quite cumbersome. In fact, the Laplace stencil includes the values of the neighbouring fields, so it cannot be applied to the boundary values and the domain size of the stencil calculation is reduced. This problem is solved in GT4Py by passing the domain size and the origin of the initial field as inputs when calling the stencil. In the case of the Biharmonic operator, the output field is formed by the Laplace operator of a temporary field, which itself is created by the Laplace operator of the input field. In GT4Py, both Laplace calculations can be performed in the same stencil, even if they are not applied to the same domain size. By simply defining the domain size of the output field, GT4Py automatically determines the required domain size for temporary fields and optimizes the calculation steps in the background. We conclude that implementing simple operators in GT4Py is generally straightforward and causes the fewest problems compared to the other languages.

Bohrium

In contrast to GT4Py, Bohrium[8] is general purpose drop-in replacement for NumPy. It provides array accelerations for the entire NumPy API and can target both multicore CPUs with OpenMP or GPUs using CUDA or OpenCL. Bohrium uses lazy evaluation, loop fusion and just-in-time (JIT) compilation to accelerate array computations. Because Bohrium does not implement the entire NumPy API, background conversions from Bohrium to NumPy can happen. If a C (and possibly CUDA) compiler is present, the installation is as simple as installing any

other pip package. The choice of the backend (CUDA, OpenCL or OpenMP) and printing of profiling or debugging information can all be done with terminal environment variables. Whilst the installation and implementation of Bohrium was by far the easiest of all the libraries, we found that its optimizations a had little positive effect on the speedup of our stencils. Due to JIT and lazy evaluation, we found it difficult to accurately time the computation of the stencil. When profiling the benchmarks with Bohrium, we found that the library correctly identified the two computational kernels and compiled them to C or CUDA code. These kernels compiled once and then ran in each iteration of our timings. For system sizes greater than 2^{12}, Bohrium causes CUDA to run out of memory. This hinders us from seeing how the accelerated computations scale at the same sizes as GT4Py and JAX. Analysis of the Bohrium program using the internal profiling system show that the cost of transfer between GPU and CPU still dominate the execution at these system sizes.

Legate

Legate [3][2] is another general-purpose drop-in replacement for NumPy. Whilst Bohrium focuses on single node performance, Legate aims to bring good weak scaling of NumPy applications to distributed, accelerated systems. It implements the NumPy API on top of a data-centric parallel programming system, named Legion [4]. The Legion runtime is designed for large HPC applications allowing it to perform well on both large GPU and CPU clusters. By decoupling the specification of the computation from the machine-dependent implementation, Legate and Legion also allows software to be easily ported to new architectures and hardware.

Legate NumPy is still in the early stages of development. In its current state, Legate NumPy must be built from source to run on large systems. This involves installing the PyArrow and CFFI packages, compiling the Legate core program with the necessary GASnet[5] implementation and linking the library to CUDA or OpenBLAS. When trying to run our stencils with Legate, we came across multiple problems, including trouble with the GASnet setup, linking of Legate NumPy and Legate Core, compilation and use of larger arrays. Because we were not able to run Legate NumPy for the relevant problem sizes, we did not include this library in the performance analysis.

JAX

Since Machine Learning is currently being used by almost every branch of industry, it is not surprising that millions or even billions are being invested in developing highly optimized and scalable ML libraries in almost every popular programming language. The most popular being Python, as it allows a high-level / declarative programming style.

A recently developed library, called JAX [6], seeks to combine the advantages of the most popular ML libraries into a framework which can directly be applied to existing NumPy / Python code. It exposes a wrapper around the existing NumPy eco-system of functions and classes, which is accessible through `jax.numpy`. This allows to easily port almost every existing NumPy code by simply changing the import statement. By exposing its own NumPy wrapper, it can take care of automatic JIT compilation, automatic vectorization and calling specialized XLA computational kernels on the most Linear Algebra manipulations. Custom functions can also be converted into fast XLA-kernels by using the function API `jit`. Similarly, there exists one for vectorization (`vmap`) or mapping a computation to multiple GPUs (`pmap`). An important thing to note is

that the arrays created by JAX are immutable, thus one might have to alter existing code, which might make it less readable (as it did in our case). This usually entails a reformulation of the mathematical operations applied to matrices / vectors. Being an ML library, it also offers automatic differentiation (forward and backward). If there is no call to compute the gradient, this functionality will not incur runtime overhead.

As an example, for the simple 2D Laplace operator from Eq. 2.2 in 2 dimensions it was necessary to create 4 separate arrays, each shifted 1 element along each dimension, to be able to mimic the stencil computation as a matrix operation. To achieve this for variable operators can result in quite some programming overhead for the user when porting a full application. We suppose that it is possible that there exist operations which ask for an even more complex transformation of the computational logic. Although this is the recommended way to achieve this [1], we suppose it can cause memory issues which were not present in the initial code (i.e. storing these additional arrays results in an at least 4 times higher memory consumption). We found that applying the `jit` directive only had small runtime benefit on our kernel computations, whereas the `vmap` directive didn't have one at all. This might be due to the matrix operations, that we call during the kernel computation, already being automatically vectorized by XLA. It must be said that we suspect not having been able to exploit the full potential JAX offers as the authors have not been previously exposed to it.

Hardware

All experiments were carried out on hybrid nodes of the CSCS (Swiss National Supercomputing Centre). Such a node consists of an Intel Xeon CPU E5-2690 v3 clocked at 2.60GHz (30MB LL-cache) that has access to a NVIDIA Tesla P100 with 16GB of memory.

Chapter 3

Results and Discussion

3.1 Runtime Analysis

In this chapter we perform a runtime analysis of the considered libraries for the Laplacian and Biharmonic stencil on a 2D and 3D grid.

Experiments on the CPU

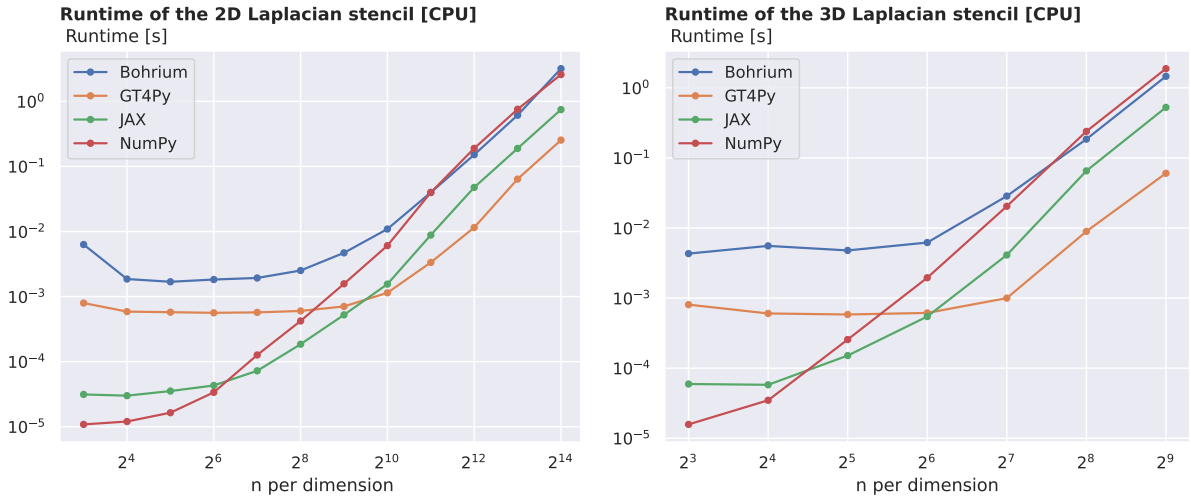

Figure 3.1: Runtime comparison for the Laplacian stencil for 2 and 3 dimensional data.

When looking at the plots in Figures 3.1 and 3.2 it is surprising that for large n, the scaling behaviour of all libraries show similar algebraic runtime behaviour. Additionally, it can be said that the order of the runtimes the libraries achieve for the largest n seems to be independent of the applied stencil type as well as the dimensionality of the grid.

GT4Py clearly outperforms all other libraries from a specific grid size onwards (i.e. $n \approx 2^9$ in 2D and $n \approx 2^6$ in 3D for the Laplacian stencil in Figure 3.1, thus when the total number of grid points is $\approx 2^{18}$). Similar behaviour can be observed for the Biharmonic stencil in Figure 3.2.

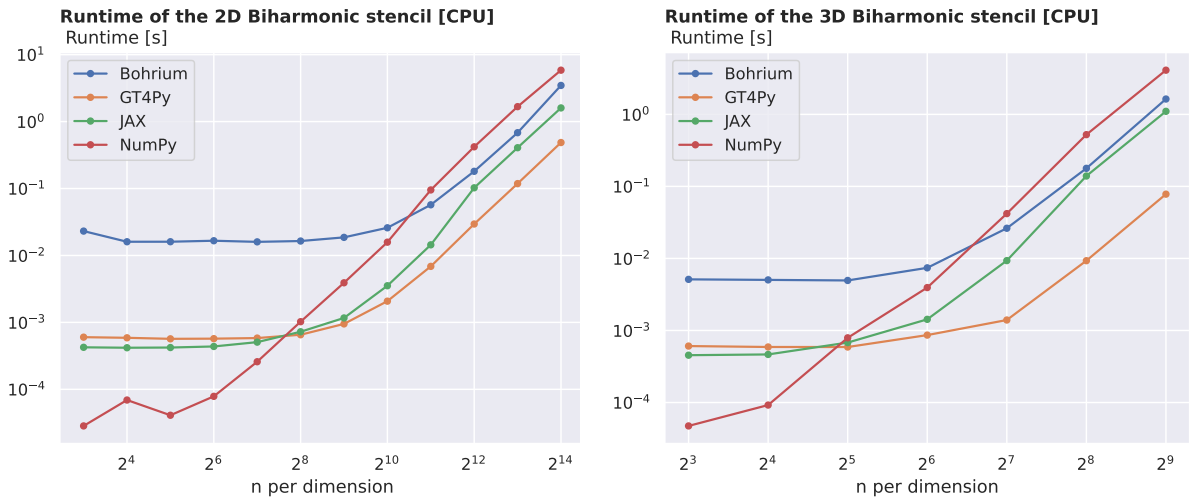

Figure 3.2: Runtime comparison for the Biharmonic stencil for 2 and 3 dimensional data.

As this holds for both the 2D and 3D case, it can be said that GT4Py handles optimizations in 3D domains just as good as in the 2D domains. This is not a given. Observing the runtimes GT4Py and JAX for small n one sees that for the Laplacian stencil GT4Py exhibits quite some overhead, resulting in much higher runtime compared to JAX. For the Biharmonic case both are very close, as there the computational cost is much larger relative to the overhead created by GT4Py. Bohrium is only able to reach NumPy's runtime for very large n. For the Biharmonic stencil it even achieves to be slightly better.

Experiments on the GPU

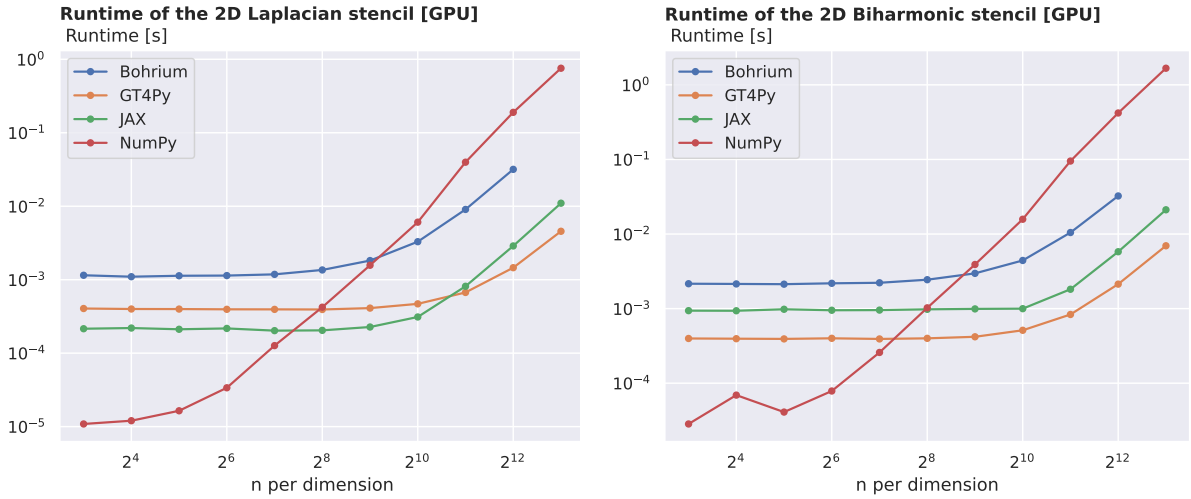

Figure 3.3: Comparison of runtimes for the Laplace operator in 2 dimensions run on the GPU. Note that NumPy is run on the CPU.

Figure 3.3 shows the runtimes of the two-dimensional Laplace and Biharmonic operators on the GPU. For reference, we have also added the runtime data for the NumPy implementation. Note

that Bohrium has problems with GPU memory for larger system sizes, so the plots only include data up to a system size of $n = 2^{12}$ per dimension. We again can observe similar behaviour of JAX and GT4Py with GT4Py performing the best for large n. Both implementations show constant run time up to the system size of $n = 2^{10}$, indicating a stronger influence of the overhead. This can be explained by the additional memory transfer needed by the GPU. The Biharmonic implementation of JAX and Bohrium produces a larger overhead compared to the Laplacian implementation, indicating more complex optimizations and precomputation. This is not the case for GT4Py.

3.2 Performance Analysis

In this section we evaluate the performance (P) of the libraries by using the following definition:

$$\mathrm{P} = \frac{\mathrm{FLOPs}}{\mathrm{runtime}} \tag{3.1}$$

To calculate the number of FLOPs, we considered the number of operations in the stencil ($N_{\mathrm{neighbours}}+1$ for Laplacian) multiplied by the number of grid points, according to the following calculation:

$$\mathrm{FLOPs}^{\mathrm{Lap}} = (N_{\mathrm{neighbours}} + 1) \cdot (N - h)^d \tag{3.2}$$

$$\mathrm{FLOPs}^{\mathrm{Bih}} = 2 \cdot (N_{\mathrm{neighbours}} + 1) \cdot (N - h)^d \tag{3.3}$$

d corresponds to the dimension of the system and h to the number of halo boundaries, which we set to $h = 4$. The number of neighbours $N_{\mathrm{neighbours}}$ in two dimensions is 4 and for three dimension is 6.

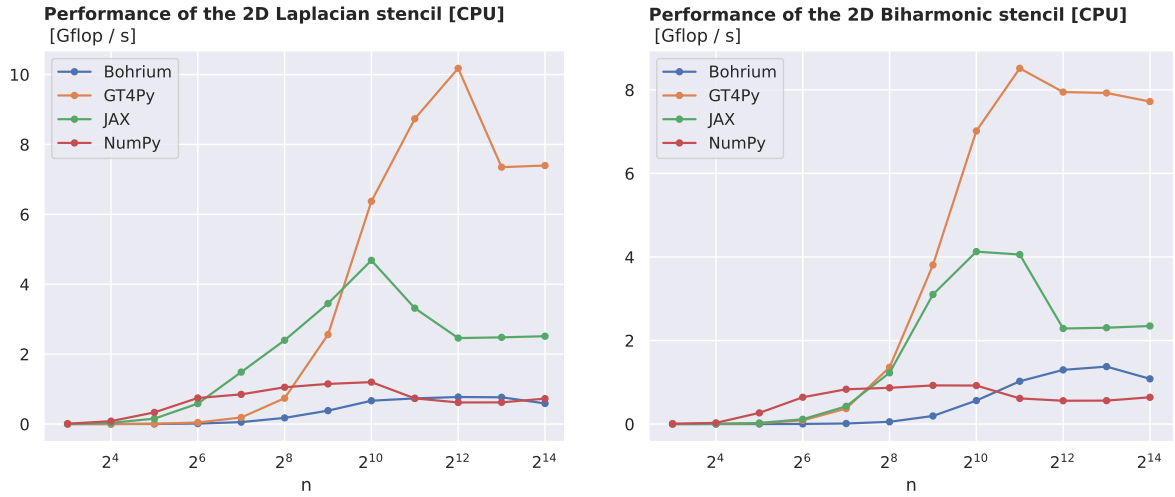

Figure 3.4: Approximate performance normalized by the respective kernel FLOP count in 2 dimensions.

In the plots of Figure 3.4 we can first observe an initial increase followed by a decrease in performance of the JAX and GT4Py implementations. This is possibly due to the fact that the problem no longer entirely fits into L3-Cache ($2^{22} \times 8\mathrm{Byte} > 30\mathrm{MB}$) and therefore increases the

number of transfers to the main memory. In the Figure on the left the JAX implementation experiences the drop sooner than the GT4Py implementation, as it needs to allocate more memory for the additional 4 grids needed for the full stencil computation (see 2.2).

The operational intensity of the Biharmonic operator is higher compared to the Laplace operator, meaning that more FLOPs per bytes are transferred. This might lead to the softer descend of the GT4Py curve we observe in Figure 3.4 on the right hand side. To our surprise, the performance of the Biharmonic stencil does not outperform the Laplace operator results. Since the biharmonic operator corresponds to the twice-applied Laplace operator, the number of FLOPs is also twice as high. We assumed that our calculations are primarily memory-bound, so that the runtime depends primarily on the intensity of the operation. Therefore, given the same memory footprint, we predicted that the performance of the Biharmonic operator could reach twice the performance of the Laplace stencil. The reason that the expected speed-up does not occur could be either that the libraries are not able to fully exploit the increased operation intensity or that the problem is already compute-bound.

Chapter 4

Conclusion

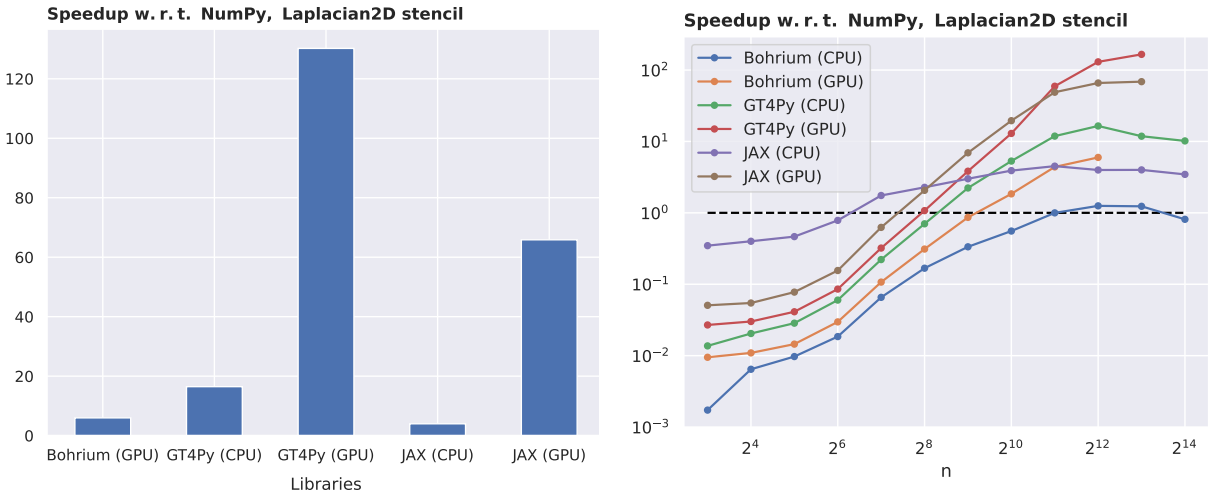

Figure 4.1: Speedup comparison with respect to the NumPy implementation for $n = 2^{13}$ (left) and for varying n (right) on a log-log plot (the dashed line represents the NumPy implementation).

In this report we have carried out a detailed comparison between GT4Py, a DSL for weather and climate simulation code, and several accelerated NumPy extensions. To enable this comparison, we implemented two computational stencils (both for 2D and 3D domains) for all libraries considered, enabling experiments on a CPU and GPU level. We have used runtime as well as approximate performance measurements up to a maximum grid side length of $n = 2^{14}$.

As we expected, GT4Py outperformed all other contestants on the CPU as well as on the GPU for large grid sizes. We summarize our findings and show the speedup of each library compared to the NumPy implementation in Figure 4.1. For small system sizes the overhead that GT4Py creates when generating the stencils, dominates the runtime and thus JAX is the preferred library due to its efficient linear algebra subroutines. Overall GT4Py has maximally a 16x speedup on the CPU with respect to the NumPy implementation and a 130x speedup on the GPU. Optimizing our kernel implementations with Bohrium had little effect on the runtime, though its main advantage being its easy usage which requires no modification to existing NumPy code and therefore might be a viable option in coming years.

Our analysis showed that a well written weather and climate DSL such as GT4Py is the optimal solution for the situations tested in this report but projects such as Legate show potential for more general purpose, highly optimized HPC DSLs designed to for exascale computing and beyond. For GT4Py to remain relevant, it must also strive to enable the user to easily deploy high-level code at similar scales.

Bibliography

[1] Skye Wanderman-Milne et al. *Solving the wave equation on cloud TPUs*. Dec. 2020. URL: https://github.com/google/jax/blob/main/cloud_tpu_colabs/Wave_Equation.ipynb.

[2] Michael Bauer and Michael Garland. *Legate*. URL: https://github.com/nv-legate/legate.core.

[3] Michael Bauer and Michael Garland. "Legate NumPy: Accelerated and Distributed Array Computing". In: SC '19. Denver, Colorado: Association for Computing Machinery, 2019. ISBN: 9781450362290. DOI: 10.1145/3295500.3356175. URL: https://doi.org/10.1145/3295500.3356175.

[4] Michael Bauer et al. "Legion: Expressing locality and independence with logical regions". In: *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 2012, pp. 1–11. DOI: 10.1109/SC.2012.71.

[5] Dan Bonachea and Paul H. Hargrove. "GASNet-EX: A High-Performance, Portable Communication Library for Exascale". In: *Languages and Compilers for Parallel Computing*. Ed. by Mary Hall and Hari Sundar. Cham: Springer International Publishing, 2019, pp. 138–158. ISBN: 978-3-030-34627-0.

[6] James Bradbury et al. *JAX: composable transformations of Python+NumPy programs*. Version 0.2.5. 2018. URL: http://github.com/google/jax.

[7] *GT4Py GitHub Repository*. URL: https://github.com/GridTools/gt4py.

[8] Mads Kristensen et al. "Bohrium: Unmodified NumPy Code on CPU, GPU and Cluster". In: Nov. 2013.

[9] S. Piasko. "Implementation of the Noah Land Surface Model Using a Domain-Specific Language". Semester Thesis. 2018.