# Performance and usability of Numba, GT4Py and CuPy for stencil computations

**Jan Wüthrich, Nina Horat and Laura Endres**

**HPC4WC block course June 2020: Project Report**
**Supervision: Stefano Ubbiali**

August 7, 2020

**High-level programming techniques to speed up Python, such as Numba, GT4Py and CuPy, become more and more popular. This project investigates the performance and usability of these modules for a range of different stencils and compares it to the performance of simple NumPy code. Numba, GT4Py and CuPy significantly boost the performance of the stencil computations compared to NumPy. The Numba njit decorator combined with the option parallel=True is a convenient and effective way to speed-up existing Python code. Numba and GT4Py show similar performance on CPU, whereas GT4Py outperforms Numba on GPU. All backends running on GPU are faster than the CPU backends. The bandwidth utilization between CPU and GPU increases with the size of the input field and peaks at around 35%.**

## Contents



## 1 Introduction

Compute-intensive tasks such as running complex numerical models or using statistical learning techniques on large data sets become more and more prevalent in scientific research and in the private industry. Often, the code for these tasks is written and maintained not by computer scientists but by field experts, which do not necessarily have knowledge of compiled languages such as C or Fortran. Therefore, several Python modules that allow to easily speed-up existing Python code have emerged in the past years.

In this project, the performance and usability of the Python modules Numba, GT4Py, Cupy and NumPy are investigated. For Numba, the decorators `@njit`, `@stencil`, and `@cuda.jit` are tested together with the `parallelize` option. For GT4Py on the other hand, the backends `gtmc`, `gtx867`,

and `gtcuda` are examined. Both, Numba and GT4Py, offer the opportunity to run code on CPU and GPU. The performance of Numba and GT4Py on GPU is also compared to the performance of CuPy. All these different backends are applied to a set of stencils that should cover the most important use cases. Their performance is compared to the performance of ordinary vectorized NumPy code.

The performance of the different modules is tested on Piz Daint, a supercomputer at the Swiss National Supercomputing Centre (Swiss National Supercomputing Centre, 2020), using a single node with a CPU and a GPU.

# 2 Methods

## 2.1 Mathematical formulation of stencils

Stencils apply a fixed computational pattern to every single grid cell of a predefined field. They are a very convenient approach for fluid-dynamical problem sets and therefore frequently found in climate and weather models.

In the present work we chose three stencils to investigate the potential and usability of a number of modules to speed up Python code: the Laplacian, the Laplacian-of-Laplacian (Lapoflap) and fused-multiply-add (FMA). All stencils are computed for a three-dimensional input field with the same halo region.

The Laplacian stencil implements the Laplace operator discretized with centred differences in space. This stencil is e.g. used to simulate diffusion. Three different versions of the Laplcacian are implemented mimicking a 1D, 2D or 3D Laplace operator. The 2D Laplace operator for example can be expressed as

$$\begin{aligned} out_{i,j,k} = &-4.0 * in_{i,j,k} + \\ &+ in_{i-1,j,k} + in_{i+1,j,k} + \\ &+ in_{i,j-1,k} + in_{i,j-1,k} \end{aligned} \tag{1}$$

where *in* and *out* are two 3D fields of the same size.

Consequently the Lapoflap stencil can be executed by calling the Laplacian stencil twice. However, the first call of the Laplcacian has to be executed on a larger domain than the second one and thus this stencil requires a careful handling of the halo region.

Finally, the FMA stencil is the only point-wise stencil investigated in this project and performs a fused multiply add operation:

$$out_{i,j,k} = in1_{i,j,k} + (in2_{i,j,k} * in3_{i,j,k}) \tag{2}$$

where *in1, in2, in3* and *in4* represent predefined 3D fields.

## 2.2 Structure of the Code

Our project code consists of several Python modules that contain the stencils and additional helper functions. Dedicated `main.py` files take care of the validation of the stencils, the time measurement of the stencil computation and the measurement of the bandwidth between CPU and GPU. All these `main.py` files use the `click` interface and thus can be run from the command line with a certain set of input parameters such as stencil name, field size, and backend.

The validation of the stencils is handled by `stencil_main_validation.py`. If this file is called with `create_field = TRUE` and `backend = numpy`, a new input field is created and saved in the folder `testfields`. Moreover, the specified NumPy stencil will be applied on this input field and the resulting output field will also be stored in `testfields`. Now, input and output fields serve as a baseline for debugging the Numba and GT4Py stencils.

All stencil computations are timed inside `stencil_main_performance.py`. Before the actual time measurement, each stencil is compiled (if necessary) and run once to avoid large initial overheads in the timing. For the timing of the GPU backends, CPU and GPU are first synchronized. The chosen stencil is executed and timed `num_iter`- times. The average and standard deviation of the stencil execution is saved in a pickle file in the folder `eval`.

In `main_bandwidth.py` the bandwidth between CPU and GPU is measured for the backends running on GPU. It is organized in the same way as `stencil_main_performance.py` and also writes the results of the time measurements to the folder `eval`.

The final validation and the time measurement for the analysis are performed in Jupyter Notebooks. The Python module `subprocess` provides the opportunity to run `.py` files from a Jupyter Notebook and thus to loop over stencils, backends

and field sizes via the specified click options for these `.py` files.

For the following analysis, each time measurement was conducted 20 times for cubic fields with $2^3$ to $2^{10}$ points in each direction. The pandas data frames generated by `stencil_main_performance.py` are again read into the Jupyter Notebook using pickle for the final evaluation and plot generation.

## 2.3 Numba

Numba is an open source just-in-time compiler for Python functions and is able to generate optimized machine code on the fly to speed up computation (Lam, Pitrou, and Seibert, 2015). The package can be installed using the command `pip install numba`.

Numba works well together with NumPy and is most commonly implemented by adding a decorator to the beginning of an existing function. In the present work the decorators `@njit`, `@stencil`, and `@cuda.jit` are used.

`@njit` enforces the function to run in no Python mode, meaning without involvement of the Python interpreter. This decorator is used for the backends `numba_vector_function`, `numba_vector_decorator` and `numba_loop`. `numba_vector_{}` is equal to the NumPy version of the stencil, apart from the added decorator, while `numba_loop` is written with nested loops using the Numba object `prange` in order to enable parallelization. This yields, for example, for Laplacian1d:

```
def laplacian1d(in_field,
        out_field, num_halo=1):
    I, J, K = in_field.shape

    for i in prange(num_halo,
            I - num_halo):
        for j in prange(num_halo,
                    J - num_halo):
            for k in prange(num_halo,
                        K - num_halo):
                out_field[i, j, k] = (
                    -2.0 * in_field[i, j, k]
                    + in_field[i - 1, j, k]
                    + in_field[i + 1, j, k]
                    )
```

Numba further provides the opportunity to parallelize the computations with the option `parallel` inside the njit command. To test the `parallel` option also in `numba_vector_function`, `@njit` was implemented not as decorator but as a function in `main_performance.py` and `main_validation.py`. This way the stencil can be computed with

```
stencil = njit(stencil,
            parallel=numba_parallel)
```

The backend `numba_vector_decorator` serves as a control whether the call by function instead of by decorator has any impact on performance.

The decorator `@stencil` is used in the backend `numba_stencil`. Within the function definition, a kernel function is defined that contains the stencil computation. This kernel function is called by the Numba stencil function that transforms it into complete but ordinary Python code. The `neighborhood` keyword can be used to provide the Numba stencil function with information on the halo.

```
from numba import stencil
def laplacian1d(in_field,
        out_field, num_halo):

    def laplacian1d_kernel(in_field):
        return -2.0 * in_field[0, 0, 0]
            + in_field[-1, 0, 0]
            + in_field[+1, 0, 0]

    out_field = stencil(
        laplacian1d_kernel,
        neighborhood=(
            (-num_halo, num_halo),
            (-num_halo, num_halo),
            (-num_halo, num_halo),
        ),
    )(in_field, out=out_field)
```

Finally, the backend `numba_cuda` performs the computation on the GPU using the decorator @cuda.jit. It is worth noting that to call a Numba Cuda function, additionally to the arguments also the threads per block and blocks per grid must be given. In this project the threads per block are set to (8,8,8). The Cuda function of Numba has to identify the active thread address with `cuda.grid` and limit the calculation to the constraints of the field size:

```
@cuda.jit
def laplacian1d(in_field,
        out_field, num_halo):
  i, j, k = cuda.grid(3)
  if i>=num_halo and j>=num_halo and
        k>=num_halo and i <
        in_field.shape[0]-num_halo and
        j < in_field.shape[1]-num_halo and
        k < in_field.shape[2]-num_halo:
    out_field[i, j, k] = (
            -2.0 * in_field[i, j, k]
                + in_field[i-1, j, k]
```

```
            + in_field[i+1, j, k])
```

The numba cuda algorithm is set up to copy the data from CPU to GPU on the fly. However to compare different GPU backends it can be useful to save the array already on the device with the command `cuda.to_device(field)`.

## 2.4 GT4Py

GT4Py (GridTools for Python), which is currently under development (`https://github.com/GridTools/gt4py`), is a Python library to access the GridTools (GT) framework using regular Python functions. The GT framework, a set of libraries and utilities, was developed with the goal to create performance portability, specifically for weather and climate models. The code is written in an generic form and then optimized for the given architecture at compile time. Pattern of stencil and stencil-like calculations are implemented in a Domain Specific Language (DSL) and embedded in C++. GT provides the 4 backend calculation options NumPy, gtmc, gtx86 and gtcuda (defined in `gt4py_backend` in the example code below). The NumPy backend is used as a reference and does not come with noteworthy improvements apart from a few optimisations. For calculations on a CPU the backend options gtmc and gtx86 optimize the code to the available hardware. The gtuca backend runs GT optimized code on the GPU. To use the speed up of GT in Python the user has to transform the fields into a GT storage class:

```
in_field = gt4py.storage.from_array(
    in_field, gt4py_backend, default_origin
)
```

The variable `default_origin` defines the halo of the field. The stencil calculation is programmed as a function and looks like this for laplacian1d :

```
def laplacian1d(
    in_field: gtscript.Field[dtype],
    out_field: gtscript.Field[dtype]
):
    with computation(PARALLEL),
        interval(...):

        out_field[0, 0, 0] = (
            -2.0 * in_field[0, 0, 0]
            + in_field[-1, 0, 0]
            + in_field[1, 0, 0]
        )
```

A standard for-loop is replaced with GT specific terminology, where `computation(PARALLEL)` represents the loop in x and y direction and `interval(...)` the loop in z direction. The split between the horizontal and vertical direction allows for optimizations in climate and weather models. Compared to normal Python code the location of the stencil points are given in relative position to the calculated field point. Before calling the function it has to be translated into a specific GT function with:

```
stencil = eval(
    f"stencils_gt4py.{stencil_name}"
)
stencil = gt4py.gtscript.stencil(
    gt4py_backend, stencil
)
```

This function can then be called like a normal function. Should further operation require it, the GT fields can be switched to NumPy fields using:

```
out_field = np.asarray(out_field)
```

## 2.5 CuPy

CuPy is a Python module that provides a simple way to transform NumPy code into GPU supporting code (Preferred Networks, Inc., 2020). Starting off with a set of NumPy stencils, only the input field has to be converted into a CuPy array before feeding it to the plain Python stencils with:

```
    in_field = cp.array(in_field)
```

This small adjustment suffices to run the stencils on GPU. After the computation of the stencils, the CuPy array has to be converted back into a NumPy array before the validation by:

```
    out_field = cp.asnumpy(out_field)
```

## 2.6 Measurement of the bandwidth between CPU and GPU

The backends Numba Cuda, Cupy, and GT4Py run on GPU and thus data has to be exchanged between CPU and GPU. While the memory allocation on GPU and the data transfer from CPU to GPU is handled separately in GT4Py, this is not necessarily the case in CuPy and Numba Cuda. For the measurement of the bandwidth, memory on the GPU and the CPU is preallocated for all backends, and the CPU and GPU are synchronized before the data transfer. Both the transfer from CPU to GPU and the transfer back from GPU to CPU are measured.

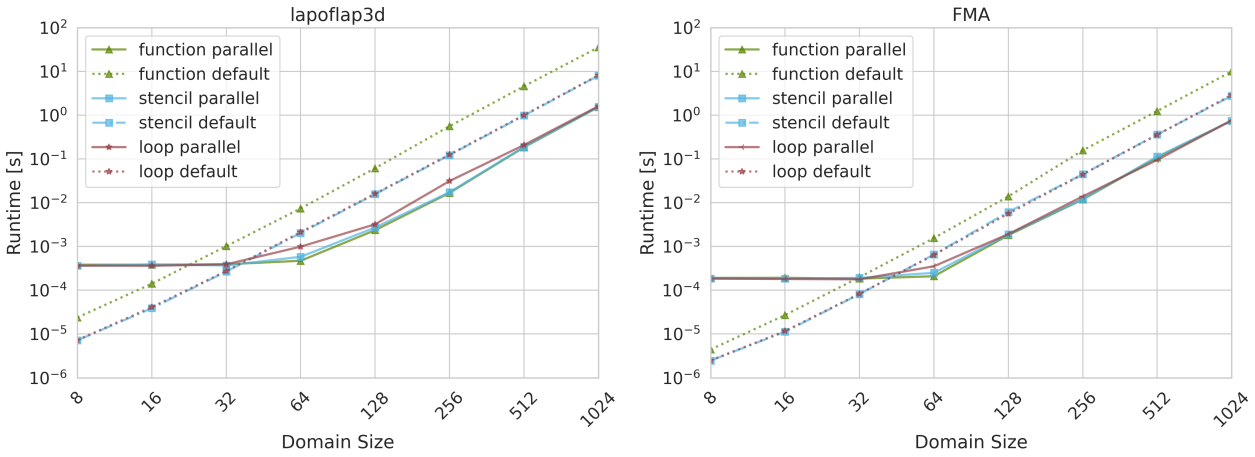


**Figure 1:** *Comparison of runtimes with Numba backends and the parallel option. Left: Lapoflap3d (most computationally expensive), Right: FMA.*

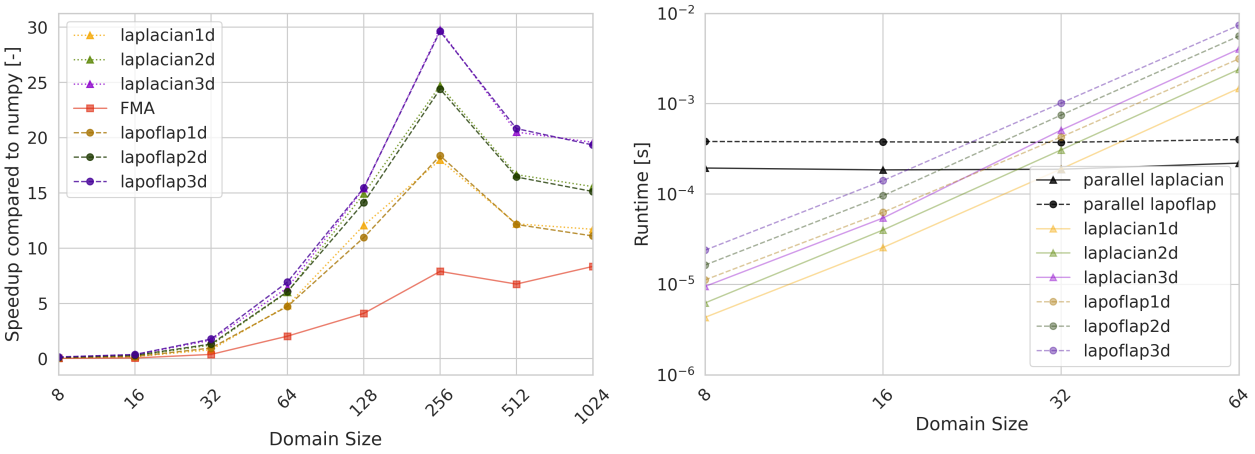


**Figure 2:** *Left: Speed-up for all stencils computed with the Numba Stencil backend and the parallel option enabled. Highest speed-up is achieved for Laplacian3d and Lapoflap3d. Right: The parallel option improves the runtime for domain sizes* $\geq 32$.

# 3 Results

## 3.1 Numba

In a first step the runtimes of the different Numba backends and the parallel option are benchmarked. In Figure 1 the stencils lapoflap3d and FMA are compared for all Numba CPU backends. Both stencils exhibit a similar pattern for the runtimes of the different backends, while the only difference is that the FMA stencil is overall shifted towards shorter runtimes. The other stencils performed similarly and therefore are not displayed here.

Generally, as expected, an exponential development of the runtime with domain size can be observed. `Numba_vector_function default` takes longer than both `numba_stencil default` and `numba_loop default` by a factor of $3.72 \pm 0.65$ (average over all stencils) regardless of domain size. Furthermore, no difference in runtime between `numba_vector_function` and `numba_vector_decorator` could be found.

If the parallel option is enabled, the runtimes in all three backends are similar (`numba_loop` might be a bit slower depending on field size) and for larger fields faster than the default option.

In order to find the fastest algorithm over all modules, the backend `numba_stencil` is selected for further analysis. The left side of Figure 2 displays the actual speedup of `numba_stencil parallel` compared to Numpy. Speedup peaks at a factor of 30 for a field size of 256 and is in each case similar for the laplacian-lapoflap pair of the same dimension. The magnified view in the right plot shows that for all stencils the runtime is improved with the parallel option enabled for domain sizes $\geq 32$.

## 3.2 Gt4Py

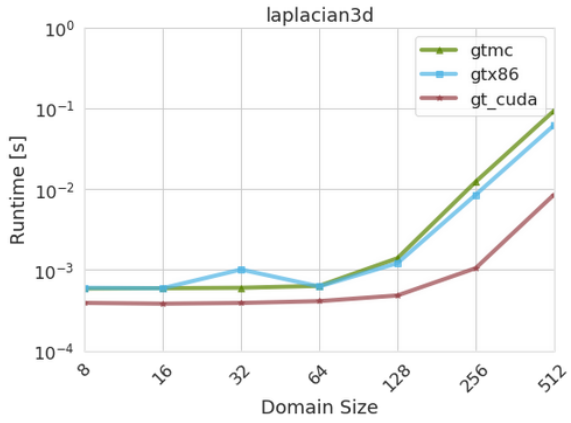


**Figure 3:** *Performance in seconds (y-axis) of stencil laplacian3d for the three GT methods $gtmc$ (green), $gtx86$ (blue) and $gtcuda$ (red) for domain sizes from 8 to 512 (x-axis).*

Figure 3 shows the calculation time for the laplacian3d stencil for different domain sizes and the three GT backend methods. For all domain sizes the `gtcuda` method, which runs on GPU is the fastest. While the two CPU methods `gtmc` and `gtx86` performed for most stencils and domain sizes comparably similar, there are a few instances, for example in Figure 3 for a domain size of 32, where there is a noticeable difference. These differences do not occur consistently when the timing calculations are rerun.

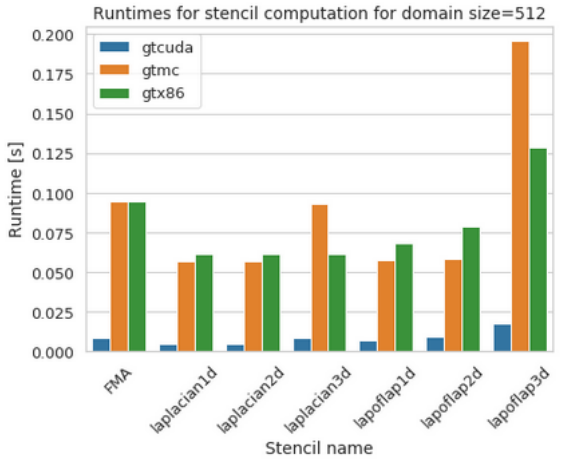


**Figure 4:** *Performance in seconds (y-axis) for the three GT methods $gtmc$ (orange), $gtx86$ (green) and $gtcuda$ (blue) for all stencil (x-axis).*

Figure 4 shows the calculation time for all stencils for a fixed domain size of 512 and all three GT backend methods. Over all stencils, the GPU backend (`gtcuda`) is consistently around one magnitude faster than the CPU backends (`gtmc` and `gtx86`). The two CPU backends perform similar and differences, for example in Figure 4 for the stencil laplacian3d, are not consistent through reruns of the timing algorithm.

## 3.3 GPU

Three of the implemented backends run on GPU: Numba_cuda uses the decorator `@cuda.jit` by Numba to perform on the GPU, Cupy is the GPU implementation of Numpy and GT4Py has the backend option `gtcuda` to run on GPU. A performance comparison for the stencils lapoflap3d and FMA can be seen in Figure 5. For the compute-intense lapoflap3d stencil GT4Py performs significantly better, for FMA CuPy is a strong contestant.

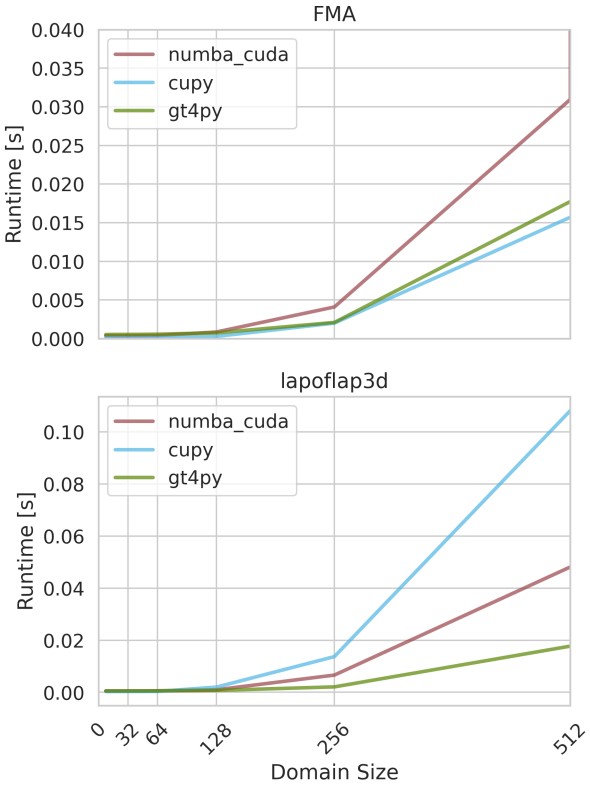


**Figure 5:** *GPU backend comparison for the FMA and the lapoflap3d stencil.*

## 3.4 Comparison between Numba, GT4Py, Cupy and NumPy

This section aims at giving an overview over the best performing backends for CPU and GPU. For Numba and GT4Py, we selected each the fastest backend on CPU and GPU. This comparison is complemented by the "base-line backend", NumPy, and its GPU counterpart, CuPy. Figure 6 shows the results for the least compute-intensive and the most compute-intensive stencil. GT4Py, Numba, and CuPy clearly offer a significant speed-up compared to NumPy and their performance is very similar for all stencils. However, there are

slight differences between the performance of the different backends for the different stencils. CuPy seems to perform well for the point-wise FMA stencil, whereas its performance is rather poor for lapoflap3d, being equal to the performance of Numba Stencil and GT4Py gtx86 on CPU. Nevertheless, running on GPU is in general advantageous for large fields since all GPU backends outperformed the CPU backends for fields larger than `128x128x128`. GT4Py gtcuda shows a strong performance for the compute-intensive lapoflap3d stencil.

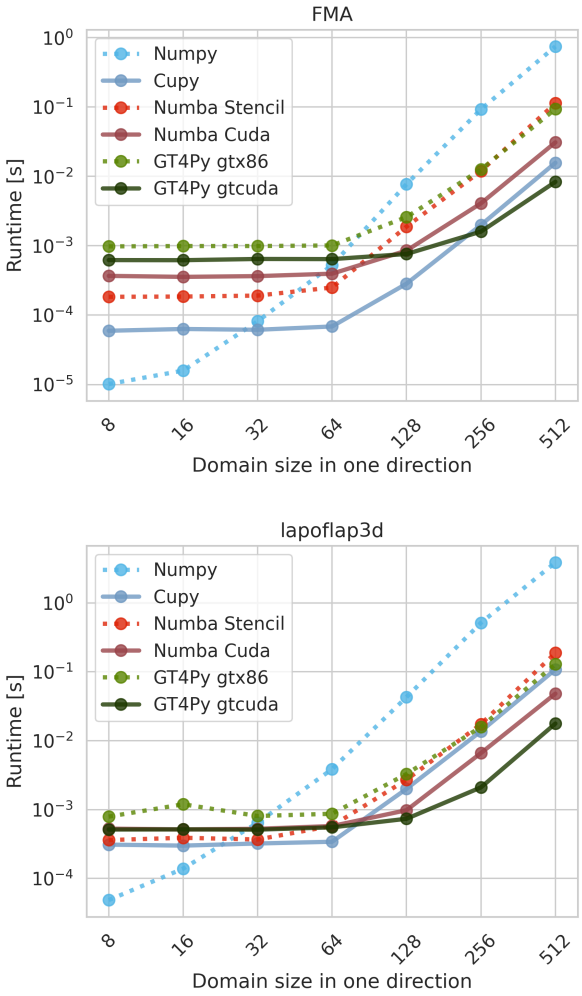


**Figure 6:** *Runtimes for the FMA and the lapoflap3d stencil with selected backends. The performance of backends running on CPU is indicated by dotted lines and the GPU backends are depicted with solid lines.*

## 3.5 Bandwidth between CPU and GPU

Several of the investigated backends run on GPU. Therefore, the input data must be transferred from CPU to GPU before the computation of the stencils and the output must be returned back to the CPU afterwards. On Piz Daint, the peak bandwidth between CPU and GPU amounts to 32 GB per second in each direction (Fuhrer, 2020). Figure 7 shows how effectively the different backends use the available bandwidth for data transfers between CPU and GPU. For each backend, the CPU-to-GPU and GPU-to-CPU data transfers achieve very similar bandwidth utilization. Generally, the bandwidth utilization becomes more efficient for increasing sizes of the input data regardless of the used backend. However, the utilization only slightly improves for domain sizes larger than `128x128x128`. The maximal bandwidth utilization lies between 29 and 39 %, which corresponds to a bandwidth of `9 - 12 GB/s`. GT4Py achieves the highest bandwidth utilization for all domain sizes. CuPy and Numba Cuda are less efficient than GT4Py, with Cupy outperforming Numba Cuda for small domain sizes.

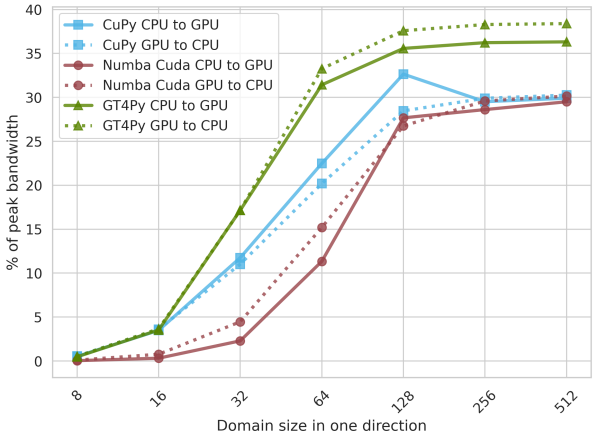


**Figure 7:** *Bandwidth for data transfer between CPU and GPU for the backends CuPy, GT4Py, Numba Cuda. The solid lines depict the data transfer from CPU to GPU and the dotted lines depict the data transfer from GPU back to CPU.*

# 4 Discussion

## 4.1 Usability analysis

### 4.1.1 Numba

Getting into Numba was comparably smooth. It is easily accessible just by installing the package via pip and advertised with the slogan *Numba makes Python fast* it is self-explanatory why we should care about Numba.

The documentation on http://numba.pydata.org/ is well structured and illustrative, especially because the chosen examples are well done and cover the relevant topics. Additionally, there is also a community already using the package: upcoming questions could be solved mostly using stackoverflow or by watching a tutorial on youtube.

While it felt almost like cheating, it was definitely a very pleasant revelation that by simply adding the decorator `@njit` and the option parallel=true already

a massive speed-up can be achieved. This is definitely a technique that we will consider using regularly beyond the scope of this work.

Using the other decorators like `@stencil` or `@cuda.jit` on the other hand needed more involvement. The decorators are quite specific and require rewriting of the original function. However, as we also could rely on the support of Stefano, occurring issues could be solved relatively quick. Maybe that is also a very important lesson of this course: Using these advanced techniques that in their detail definitely go beyond our understanding as non-computational scientists, having someone around to answer our nooby-questions and putting things in context was of great importance.

### 4.1.2 GT4Py

The GT4Py module aims to make the performance gain from the GT framework available in Python, and spares the user the struggle with machine level programming languages. Code written with the GT4Py module looks and feels like Python code and therefore is easily accessible even for users with only moderate skills in Python. However, some syntax used in the GT4Py tutorial does require advance knowledge of Python. Programming as taught in the ETH Atmosphere and Climate program is very solution oriented. Therefore, techniques and tools to write clean Python code such as exception handling or the use of decorators were new to us. Specifically the decorators were tricky to use in combination with the command `eval()` since the official GT4Py tutorial uses `@gtscript.stencil(...)`. It would help a lot if the tutorial of the GT framework was replicated for GT4Py, since there are a lot of good graphs that show what happens form a field perspective. When running into errors with `domain` and `origin`, inserting a simple field and looking at its output helped to resolve the issue. From our master program we are used to work with programs that we do not understand, however knowing what they should do with the input data. Therefore, looking at prints of fields or field slices is something we are used to.

## 4.2 Performance

The Python modules Numba, GT4Py and CuPy are an effective way to boost Python code for stencil computations. On CPU, similar performance is achieved with Numba and GT4Py. On GPU however, GT4Py outperforms Numba and, in some cases, also CuPy. It is surprising how close the performance of the CPU backends of Numba and GT4Py approaches the peformance of CuPy which runs on GPU. Moreover, GPU can only be used for field sizes smaller than `1024x1024x1024`.

Only small differences are found between the compute time of the different stencils. CuPy achieves a larger speed-up for the point-wise FMA stencil compared to the other stencils. Preferred Networks, Inc. (2020) also indicates that CuPy boosts element-wise operations more than common stencil operations. GT4Py on the other hand performs better for stencils that depend on the neighboring grid points and are as such typical for weather and climate applications.

As shown in Section 3.2, there are some performance differences between the two CPU backends `gtx86` and `gtmc` for small domain sizes. These are not consistent through reruns of the timing algorithm and become smaller when the number of iterations is increased. Therefore, we assume that the differences come from the Piz-Daint server.

The data transfer between CPU and GPU was most efficient for GT4Py. However, the differences between all GPU backends are small. The highest bandwidth utilization is achieved for the largest field size and amounts to around 30 - 40% of the peak bandwidth.

## Appendix

All time measurements are conducted on Piz Daint from the Swiss National Supercomputing Centre and as such might not be applicable to other computer architectures. The project code is available at `https://github.com/lrndrs/HPC4WC_group7`.

We would like to thank our supervisor Stefano Ubbiali for his great support and Oliver Fuhrer for organising this insightful course.

# Bibliography

Fuhrer, O. (2020). *Lecture Notes from the the block course High Performance Computing for Weather and Climate at ETHZ: GPU Programming with CuPy*. `https://github.com/ofuhrer/HPC4WC/blob/master/day4/01-GPU-programming-cupy.ipynb`. Accessed: 5. 8. 2020.

Lam, Siu Kwan, Antoine Pitrou, and Stanley Seibert (2015). "Numba: A LLVM-Based Python JIT Compiler". In: *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. LLVM '15. Austin, Texas: Association for Computing Machinery. ISBN: 9781450340052. DOI: 10.1145/2833157.2833162. URL: https://doi.org/10.1145/2833157.2833162.

Preferred Networks, Inc. (2020). *Cupy: A NumPy-compatible array library accelerated by CUDA*. `https://cupy.dev/`. Accessed: 6. 8. 2020.

Swiss National Supercomputing Centre (2020). *Piz Daint*. `https://www.cscs.ch/computers/piz-daint/`. Accessed: 6. 8. 2020.