D USYS

IAC Institute for Atmospheric and Climate Science

High Performance Computing for Weather and Climate

Stencil2D Performance Comparison: NumPy, Numba, Torch, JAX

Manuel Weber Philipp Geissmann Patricia Gribi

working on

ALPS

Advisors: Simon Adamov and Oliver Fuhrer

FS 2025

Contents

1 Introduction & Theoretical Background

1.1 Stencil Computation in High-Performance Computing (HPC)

Stencil computation is a computational pattern in which each element of a grid is updated based on a fixed pattern that takes into account its own value, neighboring elements, and possibly elements further away. In high-performance computing (HPC), stencil computations play a central role because they are widely used in applications such as solving partial differential equations, fluid dynamics, heat diffusion, and weather modeling. Since they are computationally expensive, they require careful optimization through techniques such as vectorization and parallelization.

1.2 Diffusion Equation and 2D Stencil Implementation

In the stencil code used for this work, a discretization of the Laplacian is applied to a two-dimensional grid. A five-point finite difference stencil approximates the Laplacian at grid point (i, j) as follows:

$$\nabla^2 u_{i,j} \;\approx\; \frac{1}{h^2}(u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - 4u_{i,j}) \tag{1}$$

In our implementations we set the grid spacing to $h = 1$ (non-dimensional units), so the discrete Laplacian omits the explicit $1/h^2$ factor. Applying the Laplacian twice therefore corresponds to $\Delta^2 u$ scaled by $1/h^4$ with $h = 1$. The stability parameter *alpha* is tuned for this setting. The diffusion step is implemented as:

$$u_{i,j}^{n+1} \;=\; u_{i,j}^{n} \;-\; \alpha\, \nabla^2(\nabla^2 u_{i,j}^{n}) \tag{2}$$

Thus, the Laplacian is applied twice, yielding a fourth-order diffusion scheme. This corresponds to a higher-order smoothing operator, also known as a *biharmonic diffusion* operator.

To enforce periodic boundary conditions a halo update is applied to the field. In this procedure, values are copied from one side of the domain to the halo (ghost cells) on the opposite side. In this way, boundary cells are influenced not only by their direct neighbors but also by cells across the domain.

1.3 Python-based Data Models: NumPy, Numba, Torch, JAX

Across all four data models our 4th-order stencil is ultimately *memory-bound* for realistic working sets; differences mainly stem from framework overheads and JIT warm-up rather than raw compute throughput.

NumPy provides a straightforward implementation of the stencil using vectorized array operations. It is easy to implement and efficient for smaller domains, but computations are restricted to single-core execution and quickly become memory-bandwidth limited as the domain size grows.

Numba compiles the loop nests with `@njit` to machine code, removing Python interpreter overhead. In our environment we used the *serial* variant (the parallel backend was unstable), so the results primarily show the benefit of JIT at small–medium sizes. For larger working sets, performance is worse than NumPy due to the memory-bound regime.

PyTorch (Torch) can be used as a tensor library on the CPU. For stencil computations, it handles large arrays efficiently, but the framework introduces more overhead than NumPy or Numba. Thus, for smaller domains Torch may be slower, while for larger domains its efficiency is more comparable.

JAX offers NumPy-like syntax with JIT compilation via XLA. For stencil codes, this can yield good performance on large domains by optimizing array operations, but compilation and transformation overhead are significant. This means that for small domains JAX often performs worse than simpler NumPy or Numba implementations.

1.4 Validation Strategy

To ensure the correctness of the results a visual validation method is used. The initial 2D-field is designed in a way that there is a square in the field with element values 1. Outside of this square element values are 0 (see Figure 1a). After the stencil code has been applied to the field, the square should be blurred due to diffusion performed by the stencil code (see Figure 1b).

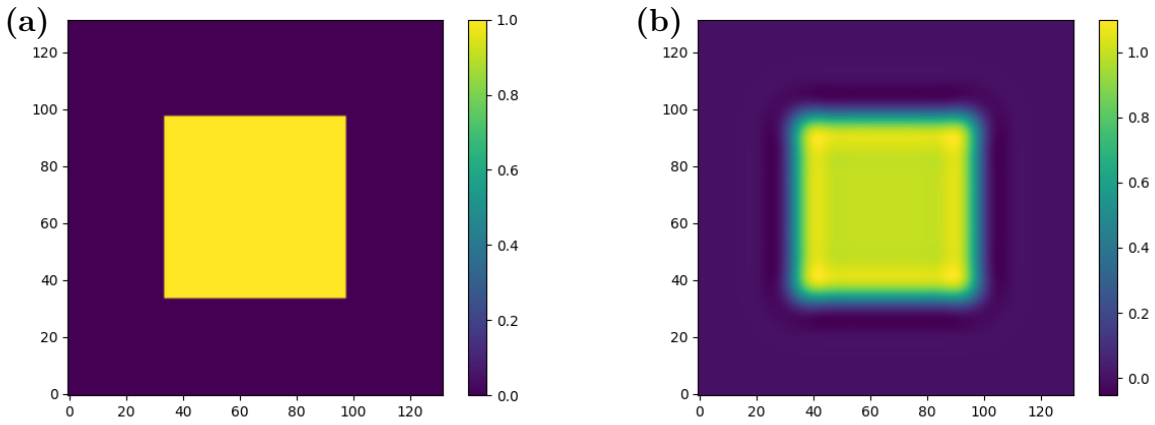

Figure 1: Validation plots used for the stencil code validation (here for the NumPy Code). (a) The input field given to the stencil code. (b) The output field after the stencil code was applied to the input field.

As long as the validation plots looked like the ones in Figure 1, we assumed that the code behaved equivalently to the initial stencil code and that no shortcomings were present in the stencil calculations across the different libraries. If the validation plots did not match the ones in Figure 1 we had a closer look at the code and fixed potential bugs.

2 Methods

The python scripts for this report are found in the hand in folder `project06_stencil2D_performance_comparison` with a readme file explaining the scripts and how to use them.

2.1 Problem and Discretization

We solve a fourth–order diffusion step on a three–dimensional field arranged as (n_z, n_y, n_x) with halo cells in x and y. Each time step applies a five–point Laplacian twice (biharmonic diffusion) on every z–slice as shown in equation (1) and (2). Because the operator is strictly 2-D in (x, y), the z-layers are independent and do not interact during this update.

Periodic boundary conditions are enforced via a halo update that copies values from one side of the domain into the ghost cells on the opposite side. Arrays are stored in C order with shape $(n_z, n_y + 2h, n_x + 2h)$, where h is the halo width.

2.2 Implementations

We implemented four backends with identical numerics and boundary handling:

- **NumPy** (`stencil2d_new.py`): pure vectorized slicing for the two Laplacians and halo updates.
- **Numba** (`stencil2d_numba_new.py`): explicit loop nest over (k, j, i) with `@njit(parallel=True, fastmath=True)`; halo and Laplacian kernels are JIT–compiled.
- **PyTorch** (`stencil2d_torch_new.py`): CPU tensors with slice-based updates; no autograd; identical halo logic.
- **JAX** (`stencil2d_jax_new.py`): `jit`–compiled kernels using `lax.fori_loop` for iteration and `.at[...].set()` for functional halo updates; CPU backend only.

All drivers perform a one–iteration warm–up prior to timing so that compilation and cache warm–up are not included in the measured section.

2.3 Hardware and Software Environment

All experiments were executed on the ETH cluster ALPS in CPU–only mode. We fixed thread counts to avoid oversubscription and library contention:

`OMP_NUM_THREADS` = k, `NUMBA_NUM_THREADS` = k, `MKL_NUM_THREADS = 1`.

For Numba stability we set `NUMBA_THREADING_LAYER=workqueue`. Runs used Python 3.12 (exact library versions are listed in the Appendix A). Torch and JAX were forced to CPU via a `--device cpu` switch.

2.4 Benchmark Protocol

To compare the runtime of the libraries we evaluated a size sweep. We varied domain sizes $n_x = n_y \in \{32, 48, 64, 96, 128, 192\}$ at $n_z = 64$ and $h = 2$, with 100 repetitions carried out for each library and domain size.
Each driver prints `''Elapsed time for work = X s''`. The harness parses this value and computes:

$$\text{rpg\_run} = \frac{t}{n_x n_y n_z}$$

reported in microseconds. For each library we plotted the distribution using box-plots (median, quartiles, and 1.5 IQR whiskers; outliers shown as points).

2.5 Validation

To ensure numerical equivalence, we plotted and compared the final fields from Numba, Torch, and JAX against the NumPy reference (see Figure 1). If the output fields looked the same as in the NumPy reference field we considered the run as correct (see Appendix B, Figure 4).

3 Results & Discussion

3.1 Raw Performance Comparison

To evaluate the performance of the three different libraries we evaluated the runtime for each library and domain size 100 times. The results are plotted in Figure 2. For all four libraries we see a decrease in runtime per grid point for increasing domain sizes. This is due to the reduced overhead per grid point with increasing domain size. However, there are also other reasons which are discussed in the next chapter.
While run time sample spread of single domain sizes of Numpy (a) and Jax (d) show a quite narrow distributions the data distribution of Torch (c) is rather wide. The single domain spread of Numba (b) and Jax (d) show many outliers in comparison to Numpy and Torch.

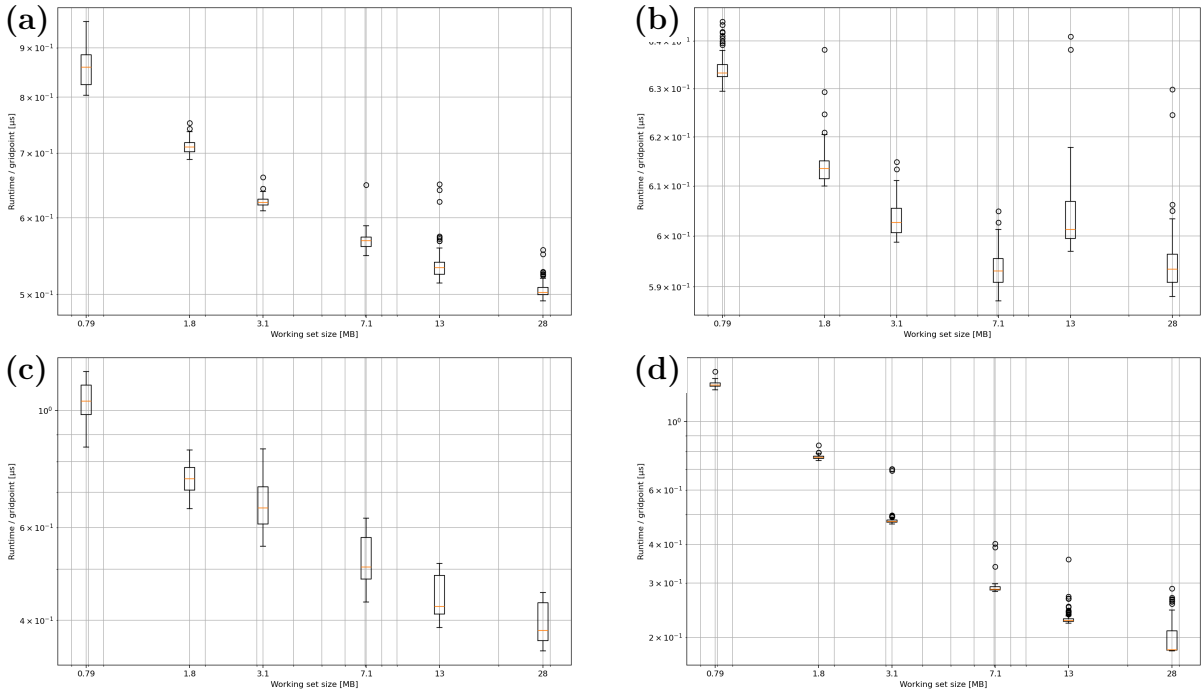

Figure 2: Stencil2D performance by framework: runtime per grid point vs. working-set size for (a) NumPy, (b) Numba, (c) Torch and (d) Jax. Working size steps are $n_x = n_y \in \{32, 48, 64, 96, 128, 192\}$.

While the run time of NumPy, Torch and Jax decreases comparably with increasing domain size, changes of Numba are one order of magnitude below the other three (compare y-axes). Further, Numba shows an average runtime increase from domain size 96x96 to 128x128.

Comparing the median runtime per grid point between the four libraries we can see in Figure 3 that the choice of the best-performing library depends on the working set size. While Jax performs worst for small domain sizes, already at sample sizes of 64x64 it outperforms the other libraries and stays the best-performing library for even larger domain sizes. As already seen in Figure 2b Numba isn't able to significantly reduce its runtime per grid point with increasing domain size. However, in comparison to the other libraries Numba is very efficient for small sample sizes. Torch and NumPy can be seen as intermediate performing libraries, that are not specialized on large or small domain sizes and therefore (in comparison to Numba and Jax) perform mediocre for all sample sizes.

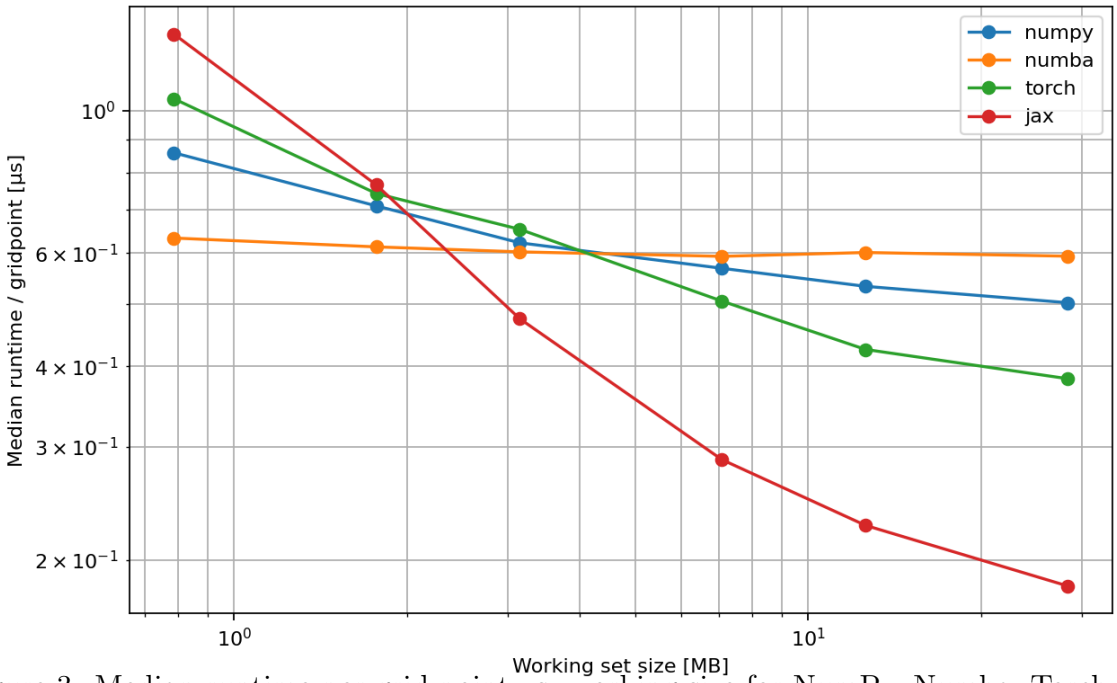

Figure 3: Median runtime per grid point vs. working size for NumPy, Numba, Torch and Jax. Working size steps are $n_x = n_y \in [32, 48, 64, 96, 128, 192]$.

3.2 Why Certain Data Models Are Faster

As figure 3 illustrates, the performance differences between NumPy, Numba, Torch and Jax can be explained by HPC effects, we have seen in the course. When running small working set sizes the runs are dominated by fixed overheads, meaning in python with library dispatch, kernel setup and copying halos. There JIT- based systems (Torch and Jax) have a disadvantage. They pay some of these cost upfront and therefore look slow with small working set sizes. As the arrays get larger these fixed costs are amortized and the raw loop throughput starts to dominate, which is why per-grid-point time drops with domain size.

A second effect is geometric. In each 2D slice we update halo cells around the border. The amount of work and time needed for this, grows with the perimeter that scales like N (the number of grid points in in the x and y direction of a 2D slice). Interior works on the other hand scales with the area N^2. Meaning For small N you spend a larger fraction of time on halos, which increases the measured microseconds per point.

Another important factor is threading. For Numba we used a single threading, while JAX and Torch use multi-threaded CPU backends by default. That's why Numba is great for tiny cases (very low overhead) but falls behind on larger grids where using many cores wins.

A reason for the good performance of JAX with larger sizes is, the fusion of the two Laplacians plus the AXPY interior update into few, cache friendly loops. So reducing memory traffic and improving locality. Since for very large grids the computation becomes mostly memory-bandwidth limited. Frameworks that move fewer bytes per update or overlap memory access with compute, end up with the lowest time per point. In our case this is JAX.

4 Conclusion

4.1 Summary of Key Findings

The performance of the different library programs is influenced by different factors resulting in different performance for changing domain sizes. Per–grid-point time decreases with size because constant costs are amortized and the surface/volume penalty of halos shrinks. For very small problems, Numba (single-threaded) is fastest due to minimal overhead. For larger (more realistic) domain sizes, JAX delivers the best CPU throughput; Torch is second, NumPy a solid baseline, and Numba (as configured) trails. Consequently, we recommend to use Numba for small domain problems, Jax for large problems and Torch and NumPy if we confront a problem where domain size changes.

4.2 Limitations of the Study

Numba's potential is understated, since we only looked at single-threaded Numba kernels, because we wanted to avoid problematic parallel backend crashes we encountered. This means our Numba baseline does not include the speedups available with @njit(parallel=True), prange, and a tuned threading layer.

Furthermore, we didn't pin CPU thread counts for Torch and JAX explicitly, which makes the comparison and reproducibility harder (speed up could result from different number of cores used and not only from different code).

Another limitation is that we only have results for one machine. Runs on other architectures could change absolute numbers and relative order, since hardware characteristics (core count, clock, cache sizes, memory type) and instruction sets vary widely across systems and influence performance of different programs.

5 Outlook

This project could be taken further by using the Numba script in parallel. We could refactor its loops with prange and evaluate i.e. study scaling vs. core counts. Another

step would be to pin threads, so we can compare the different programs better and fairer.

Furthermore other comparisons can be added, e.g. a GPU study. Therefore, we could benchmark Torch/JAX on CUDA and compare CPU-GPU crossover sizes and performance.

References

[1] O. Fuhrer, *HPC4WC Course Materials*, ETH Zürich, 2025.

[2] *NumPy Documentation*, `https://numpy.org/`

[3] *Numba Documentation*, `https://numba.pydata.org/`

[4] *Torch Documentation*, `https://docs.pytorch.org/docs/stable/index.html`

[5] *JAX Documentation*, `https://jax.readthedocs.io/`

A Library Versions

Python version: 3.12.5 (CPython, GCC 13.3.0)

Table 1: Key package versions

Package	Version
numpy	2.1.3
numba	0.61.2
torch	2.7.1+cpu
jax	0.6.2
jaxlib	0.6.2
matplotlib	3.10.3
click	8.2.1
torch_cuda	None
torch_cudnn	None

B Additional Figures

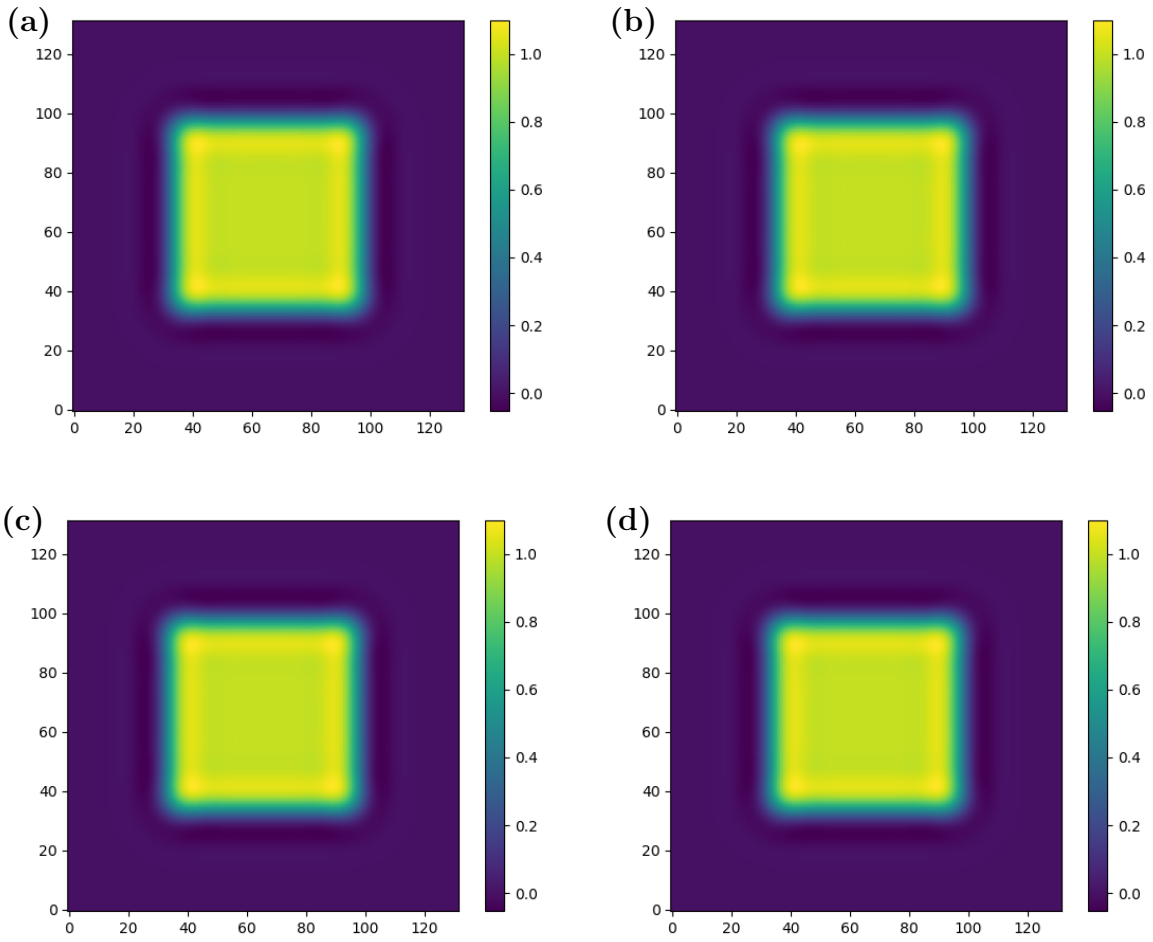

Figure 4: Output fields of (a) NumPy, (b) Numba, (c) Torch and (d) Jax for $n_x = n_y = 128$ and $n_z = 64$. All runs are considered as correct.

C Use of AI-Based Tools

In this work we used an AI assistant (ChatGPT, GPT-5 by OpenAI) to help with coding, debugging, and editing.