Optimizing Diffusion Calculations: A Performance Study of NumPy, Numba, Torch and JAX

Tobias Rahn, Hanna Kjellson, Fabrice Stefanetti

September 13, 2024

1 Abstract

In this project, the data models NumPy, Numba, Torch and JAX were used to implement the fourth order diffusion equation over a cuboid domain. The goal was to compare the data models and to determine the best data model for diffusion simulations. The code was run on one node of the supercomputer Piz Daint [1] and to compare the models, the precision (32 or 64 bit) and the number of cores (1, 6 or 12) were varied. When several cores were used, Torch had the best performance, due to its highly optimized parallelization capabilities. However, when only one core was used, JAX performed best for 32 bit precision, while Numba performed best for 64 bit precision. These models performed well, as both of them use Just In Time (JIT) compilation, allowing for additional optimization and efficient memory accessing.

2 Introduction

Diffusion is a fundamental physical process in which particles or substances spread from areas of higher concentration to areas of lower concentration. For example, diffusion is crucial in processes such as the mixing of gases or the absorption of nutrients in biological systems. To calculate diffusion on a computer, stencil programs are often used, and in this project, 4 different data models; NumPy, Numba, Torch and JAX, were used to implement such a program. The objective was to compare the performance of these data models in terms of runtime and to investigate why certain data models performed better than others.

The report is structured as follows. In Section 3, a short introduction to the physical problem and its discretization is provided. This is followed by a description of the different python data models. Next, in Section 4, the implementation procedure and the process of generating results are described. In Section 5 and

6 the results are presented and discussed so that conclusions can be drawn in section 7. Lastly, Section 8 shows how the results can be reproduced.

3 Background

3.1 Fourth Order Diffusion

Assuming some quantity u, the fourth order diffusion equation in two dimensions is described by Equation 1.

$$\frac{\partial u(x,y,t)}{\partial t} = -D\Delta(\Delta u) \tag{1}$$

D is the diffusion coefficient and Δ the Laplacian operator $\frac{\partial^2}{\partial x^2}+\frac{\partial^2}{\partial y^2}$. In this project u was defined in three dimensions, but diffusion was modelled in the horizontal. Thus, $u(x,y,t)$ in equation 1 could be replaced by $u(x,y,z,t)$ [2].

3.2 Stencil Computations for Equation 1

To discretize equation 1, the first task was to discretize the Laplacian operator. Taking a dimensionless approach and introducing $u^t_{i,j,k}$ as u at time t and position i, j, k, this was done as in equation 2.

$$\Delta u(i,j,k,t) \approx u^t_{i+1,j,k} + u^t_{i-1,j,k} - 4u^t_{i,j,k} + u^t_{i,j+1,k} + u^t_{i,j-1,k} \tag{2}$$

Applying this discretization twice and multiplying the result by $-D$ yielded an approximation of the right-hand side of equation 1. Adding this to the previous $u^{t-1}_{i,j,k}$ resulted in an estimate of $u^t_{i,j,k}$. For this update to work at the boundaries, two additional rows/columns of grid points had to be added at all sides of the horizontal domain. To update these rows and columns, periodic boundary conditions were assumed [2].

3.3 NumPy

In NumPy, the primary data structure is the `ndarray`, a multidimensional array which contains elements of the same data type. Having the same data type allows for efficient memory usage, as the `ndarray`s can be stored in contiguous memory blocks, with either row-major or column-major order. When arrays are modified, the elements are changed in place [3].

3.4 Numba

Numba also uses `ndarrays`, but translates python functions to machine code using just-in-time (JIT) compilation with LLVM (Low-Level Virtual Machine)-based optimizations. With this approach, some code is compiled at runtime, rather than before execution, allowing for further optimization of memory access and data locality. Numba can additionally automatically parallelize loops, which improves performance on multicore CPU and GPU. GPU capabilities are not standard, but can be added using CUDA [4].

3.5 Torch

In Torch, the fundamental data structure is the `tensor`, which is analogous to the `ndarray` as all elements are of the same data type and can be modified in place. The main difference is that Torch is better at parallelizing code and that it supports both CPU and GPU, with simple transferring between the two. Torch additionally has an Autograd engine, which tracks operations for gradient calculations [5].

3.6 JAX

JAX primarily uses the data structure `jax.numpy.ndarray`, which is an extension of the NumPy `ndarray`. The elements are of the same data type but instead of modifying the elements in place, new arrays are created. Like Numba, JAX can just-in-time (JIT) compile operations to optimized machine code. Instead of LLVM, Numba uses XLA (Accelerated Linear Algebra) operations to JIT compile. These operations are designed with machine learning tasks in mind and are optimized for 32 bit precision. Similarly to Torch, code can be parallelized and arrays can smoothly be moved between CPU and GPU devices [6].

4 Method

4.1 Implementation

The NumPy implementation used in this project was to a large extent based on the implementation from the course *High Performance Computing for Weather and Climate* [2]. Some updates were made to allow for performance measuring and command-line execution. For instance, parameters for domain size, the number of iterations, and the precision (32 or 64 bit) were added as CLI arguments, using the python library `click`.

To switch to Numba, the `numba.jit` decorator was used on the functions *laplacian* and *update_halo*, with parameters `parallel=True` and `fastmath=True`. For Torch and JAX, all NumPy arrays were changed to `tensors` and `jax.numpy.ndarrays` respectively. To ensure that the implementations were correct, a script was created to check the solutions, see Section 8.1 for more information.

Additional versions were created for Torch, Numba and JAX to improve performance. For Torch, this was done using convolutions for the calculation of the discretized Laplacian. However, the runtime increased, so this version was put aside. The additional implementation for Numba was done once using the `@vectorize` and `@stencil` decorators, but the implementation did no longer yield correct results, so it was not considered further. A third version of the Numba implementation involved, among other changes, passing the *update_halo* and *laplacian* functions as arguments into the *apply_diffusion* function. This approach allowed for better optimisation and inlining of these functions, which reduced overhead and improved runtime significantly. Additionally, applying `@njit(parallel=True,fastmath=True)` to *apply_diffusion* as well as using `np.ascontiguousarray` for the input arrays, contributed to further performance gains. However, due to time constraints, this improved version was not further integrated into the report. The additional implementation for JAX on the other hand improved the performance and was integrated in the analysis. It was implemented using JIT compilation on the same functions as the original Numba version and is in the rest of the report referred to as *JAX* while the previous version is referred to as *JAX base*. The attempts for Numba and Torch that were not used can be found on the GitHub of this project [7].

4.2 Testing Setup

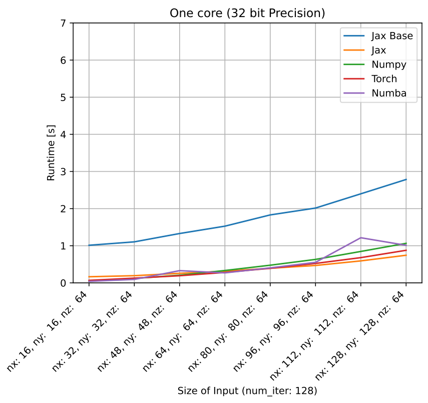
We ran the measurements on Piz Daint[8]. We used the *XC50 Compute Nodes* which are equipped with Intel Xeon E5-2690 processors that have a base frequency of 2.60GHz, 12 cores and 64GB of RAM. They are additionally equipped with an NVIDIA Tesla P100 GPU that has 16GB of VRAM. Due to some library version incompatibilities, we decided to only run our test on the CPU and to not use the available GPU. For more information on how to reproduce the results, see Section 8.2.

4.3 Plotting and Analysing

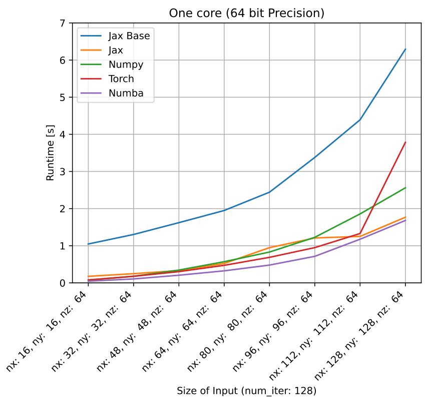
To compare the data models, plots of runtime as a function of domain size were created for all models and for 6 different cases with varying precision (32 and 64 bit) and number of cores (one, six, and twelve). Note that the domain was assumed to be a square in the horizontal and fixed in the vertical, that is `nx`=`ny` and `nz`=64.

5 Results

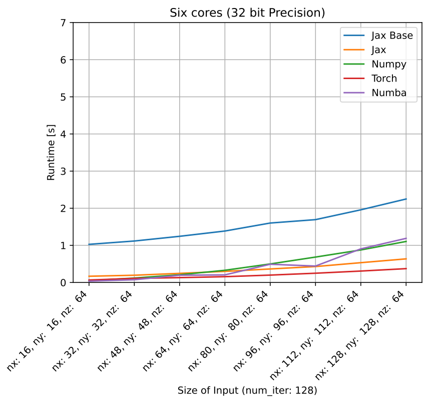
The runtime as a function of domain size for precision 32 and 64 bit and for one, six and twelve cores is presented in Figure 1. A first observation was that JAX, Torch and Numba performed better than NumPy in almost all cases. Torch had the shortest runtime for multiple cores, while JAX (beyond a certain domain size) and Numba were better for one core and 32 and 64 bit respectively. JAX base had the lowest performance in all cases. To see more clearly the impact of an increase in precision and in the number of cores on the runtime, some general behaviours are presented in table 1.



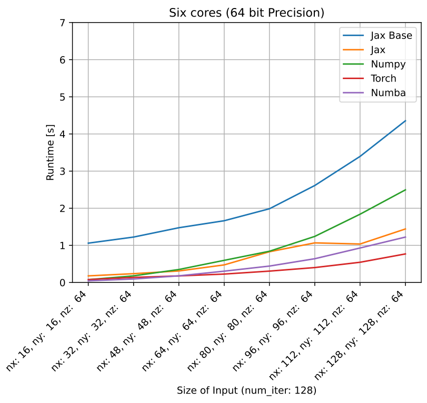
(a)



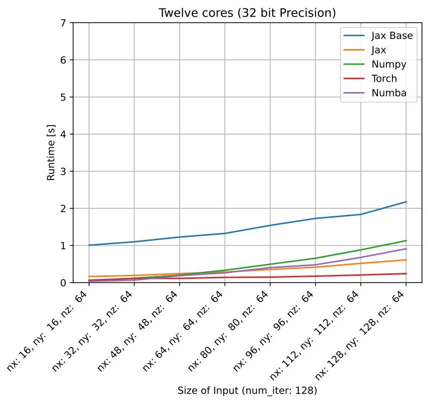
(b)



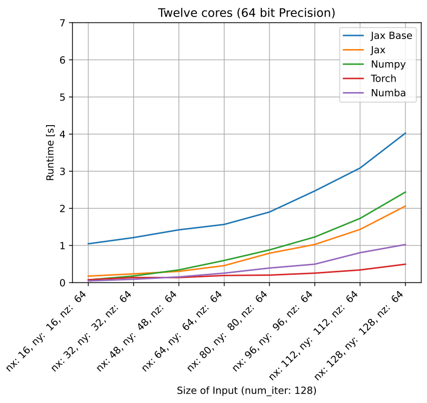
(c)



(d)



(e)



(f)

Figure 1: Runtime as a function of input size for one, six and twelve cores and precision 32 and 64 bit.

Data Model	↑ Precision	↑ # of cores (32 bit)	↑ # of cores (64 bit)
NumPy	Large increase	No difference	No difference
JAX base	Large increase	Small decrease	Small decrease
JAX	Large increase	Small decrease	Not clear
Numba	No difference	Small decrease	Small decrease
Torch	Small increase	Small decrease	Small decrease

Table 1: Runtime response to an increase in precision and to an increase in the number of cores for 32 and 64 bit respectively.

”Not clear” was assigned to JAX for an increase in the number of cores with 64 bit precision, as there was a decrease in runtime from one to six cores, while there was an increase in runtime from six to twelve cores. Another interesting aspect was that there was a large increase in runtime for Torch when the domain size was maximized, the precision was 64 bit and only one core was used. This was not taken into account when considering the general behaviour in Table 1. One last aspect that might be worth mentioning was that NumPy sometimes outperformed Numba. This often happened at points where the Numba runtime appeared to first increase and then decrease with an increase in domain size, for instance in Figure 1a. A similar behaviour could sometimes also be seen for JAX, for instance in Figure 1d.

6 Discussion

6.1 Analysis

When memory usage increases, data is moved to the L2 and L3 caches, increasing the time it takes to access it. When enough data is stored, there is a risk of cache misses, leading to use of the slow memory bus to RAM. As increasing the precision and the domain size results in a higher memory usage, the expected response to this was an increase in runtime. Additionally, the higher precision operations take longer in terms of CPU cycles.

As can be seen in Table 1, the runtimes for NumPy, JAX base and JAX, all increased when the precision was set to 64 bit. NumPy, being single-threaded, was particularly sensitive to the increased memory demands of 64-bit precision, as it could not offset these costs through parallel processing, unlike Torch, Numba, and JAX. JAX base did allow for parallelization, but the performance gains were small. As it additionally did not take advantage of JIT compilation, the performance and response to an increase in precision, were basically the same as for NumPy, but with an overhead from the use of `jax.numpy.ndarray`. For JAX, JIT compilation was used, but the XLA operations are optimized for 32 bit precision and large datasets, as JIT compilation reduces the number of memory accesses by fusing operations. Consequently, there was an additional overhead when the precision was set to 64 bit and for smaller workloads, see instance Figure 1a and 1b.

On the other hand, Numba and Torch seemed to be less sensitive to changes in precision, especially when using multiple cores. This might have been due to how the different models handle memory access, but for multiple cores it might also have been a result of the efficient parallelization in Numba and Torch. Based on Figure 1a and 1b, the Torch runtime for one core appeared to increase with an increase in precision. This might have been due to its Autograd engine, which while useful for machine learning tasks, adds overhead for stencil programs, particularly when memory access patterns do not align well with the underlying hardware. However, for multiple cores, this problem was not obvious, so parallelization seemed to play a big role for the high performance of Torch. Numba only increased marginally and mainly for the largest domain size in Figure 1a and 1b. Thus, it appeared as if Numba did not only rely on efficient parallelization, but that it also had an efficient memory access pattern. This, as Numba uses JIT compilation with LLVM-based optimizations that target CPUs and generalize well to 64 bit precision.

However, the ability of Numba's JIT to fully optimize memory accesses depends on the underlying cache hierarchy and array size, which may result in inconsistent performance depending on the data layout. This might explain why Numba sometimes performed worse than NumPy, see for example Figure 1a. Suppose it was difficult to fit the array of size (112,112,64) into the caches efficiently. Increasing the size could then, in principle, make the array more suitable for efficient cache usage, resulting in a zigzagging behaviour and in some cases a surprisingly "poor" performance. As JAX also uses JIT compilation, it was not surprising that the same behaviour could be seen for JAX, for instance in Figure 1d.

Moving on to the number of cores, it should first be noted that NumPy does not exploit parallelization, and thus, as expected, there was no improvement for NumPy when the number of cores increased. For the other models, an increase in the number of cores was in general expected to lead to a decrease in runtime. This was indeed the case for Numba, JAX base and Torch, independently of the precision and for JAX with 32 bit precision. For JAX with 64 bit precision, the interesting behaviour with an increase in runtime from six to twelve cores, could originate from the overhead that comes with parallelization. Similarly to how Numba and JAX were sometimes not able to optimize memory access efficiently enough, JAX might sometimes not have been able to parallelize efficiently enough. The increase was mainly for the two largest domain sizes, so those specific cases would not have been very compatible with the use of twelve cores.

6.2 Further investigations

In Figure 1 it is clear that JAX did respond to parallelization, but not to the same extent as expected. It might have been that parallelization in JAX was not used to its full capacity and late on in the project, the function `pmap` that would allow for further parallelization, was found for JAX. Applying this to some functions could perhaps have improved the performance.

As mentioned in Section 4.2, some attempts were made to be able to run the code on GPU. While these failed, it would have been interesting to see how this would have affected the performance of for instance Torch and JAX. Both Torch and JAX are highly optimized for GPU execution, where memory access and parallelization patterns differ significantly from CPU-based execution. This could result in much lower runtimes for large domains and higher precision workloads, which are likely to benefit from the massive parallelism of GPUs. Perhaps the use of GPU would also have improved the performance of the Torch version using convolutions.

Furthermore, attempts were made to profile the code, but while the memory profiler did yield results, they were not very useful as they only showed the total memory usage and not the use per cache. A successful, more general profiling of the code would have been helpful for interpreting the results. Lastly, it would have been interesting to see how the runtime would change if a rectangle instead of a square domain was used in the horizontal. In class [2], rectangular and square domains yielded different results, and perhaps it would have been possible to connect this to whether the data models use row- or column major ordering.1

7 Conclusion

While several areas, such as GPU performance and varying domain sizes, remain unexplored, some conclusions can be drawn from this project. Torch consistently outperformed the other models for both 32 and 64-bit precision on multiple cores, primarily due to its advanced parallelization capabilities. For a single core, JAX excelled at 32-bit precision due to its optimized JIT compilation for lower precision tasks, whereas Numba showed superior performance at 64-bit precision, benefiting from its efficient memory management and optimizations for higher precision. Hence, with one core, JIT compilation played a crucial role, as it enabled significant optimizations and efficient memory access. However, this sometimes led to unpredictable 'zigzagging' behaviours in performance, where the runtime could increase and decrease with varying domain sizes and precision levels.

8 Result Reproduction

The whole code base can be found in the Git repository [7]. Some basic instructions can be found in the respective `README.md` files.

8.1 Correctness

Correctness can be tested against the reference implementation mentioned in Section 4.1 [2]. One small adjustment needed to be done, to ensure that the results of this implementation were consistent for an even and odd number of iterations of the diffusion. Additionally, support for 32 bit vs 64 bit precision was added along with some other adaptions such that the code was compatible with the wrapper scripts we used.

The script to test correctness takes a file path as input (which needs to adhere to some guidelines specified in [9]). The script then automatically parses the filename and checks whether the respective results are already present, or generates them if necessary. We checked all our scripts for correctness with the 32 bit and 64 bit versions. To check whether the results were actually similar, a relative tolerance of 10^{-5} and an absolute tolerance of 10^{-8} were used (default parameters of `numpy.allClose()`).

8.2 Measurements

We took all our measurements on the system mentioned in Section 4.2. The measurements were taken automatically using the script `tester.py` [10]. We used the following command to run the aforementioned script in the SLURM cluster [1]:

```
srun --account=class03\
--constraint=gpu\
--partition=normal\
--nodes=1\
--ntasks-per-core=1\
--ntasks-per-node=1\
--cpus-per-task={1,6,12}\
--hint=nomultithread\
--mail-type=ALL\
--mail-user={nethz}@ethz.ch\
--time=00:15:00\
--output=./result-%j.txt\
python scripts/tester.py
```

We ran the code only on a single node and with a single task as we did not use a technique like MPI in our implementations. But we did vary the number of CPU cores available per task. The data of our runs can be found on GitHub [11]. `plain` corresponds to one CPU core per task and the others to six and twelve CPU cores per task respectively.

We used the python environment from the course GitHub [2] (python version 3.9.4) but installed some more packages listed in the file `requirements.txt` [12]. Once logged into the cluster the following steps should be performed before running the `srun` command above:

```
module load daint-gpu
git clone https://github.com/ofuhrer/HPC4WC
./setup/HPC4WC/setup/HPC4WC_setup.sh
source HPC4WC_venv/bin/activate
git clone https://github.com/ib31iat/HPC4WC-Project-13
cd git/HPC4WC-Project-13/
pip install -r requirements.txt
```

8.3 Plotting

To create plots with our data, we used the self-explanatory Jupyter Notebook [13].

References

[1] Piz Daint Cluster, https://user.cscs.ch/access/running/piz_daint/

[2] HPC4WC Github, https://github.com/ofuhrer/HPC4WC/blob/main/day1/stencil2d.py (accessed 28.06.2024)

[3] NumPy, *Array objects*, https://numpy.org/doc/stable/reference/arrays.html (accessed 10.09.2024)

[4] Numba, *A 5 minute guide to Numba*, https://numba.readthedocs.io/en/stable/user/5minguide.html(accessed 10.09.2024)

[5] PyTorch, *torch.Tensor*, https://pytorch.org/docs/stable/tensors.html(accessed 10.09.2024)

[6] Medium, *JAX vs. NumPy: Key Differences and Benefits*, https://medium.com/@harshavardhangv/jax-vs-numpy-key-differences-and-benefits-72e442bbf67f(accessed 10.09.2024)

[7] HPC4WC-Project-13, https://github.com/ib31iat/HPC4WC-Project-13

[8] Piz Daint, https://www.cscs.ch/computers/piz-daint

[9] Scripts README.md, https://github.com/ib31iat/HPC4WC-Project-13/blob/main/scripts/README.md#instructions

[10] Measurement Script, `https://github.com/ib31iat/HPC4WC-Project-13/blob/main/scripts/tester.py`

[11] Results, `https://github.com/ib31iat/HPC4WC-Project-13/tree/main/results`

[12] Python Requirements File, `https://github.com/ib31iat/HPC4WC-Project-13/blob/main/requirements.txt`

[13] Plotting Jupyter Notebook, `https://github.com/ib31iat/HPC4WC-Project-13/blob/main/scripts/plotting.ipynb`