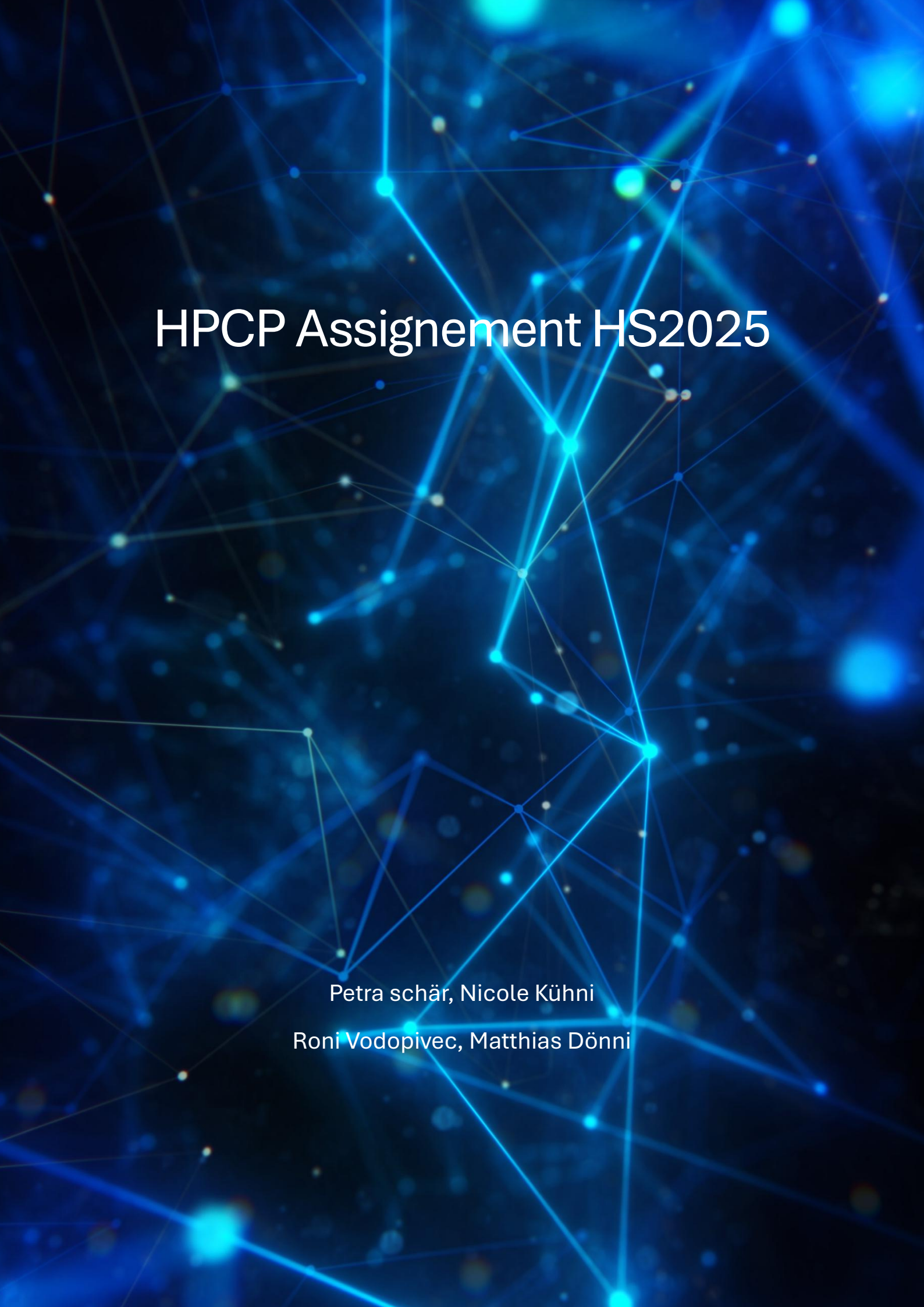


HPCP Assignement HS2025

Petra schär, Nicole Kühni

Roni Vodopivec, Matthias Dönni

Contents

1. Assignment Description 3
1.1. General approach 3
2. Workflow 4
2.1. Step 1: Profiling the “de2dem_pos” function 4
3. Optimizations 5
3.1. "dem_reg_map" with NumPy Vectorization 5
3.1.1. Performance boost 6
3.1.2. Profiling after NymPy Vectorization 7
3.1.3. Confusion and warm up 7
3.2. "dem_reg_map" with Numba / JIT 10
3.2.1. Performance boost 11
3.2.2. Profiling after Numba / Jit 11
3.2.3. Conclusion 11
3.3. "dem_reg_map" with CuPy 12
3.3.1. Result 13
3.4. "dem_reg_map" with Numba / JIT (no parallel) 14
3.4.1. Performance boost 15
3.4.2. Profiling after 15
3.4.3. Conclusion 15
3.5. Overall conclusion 16
Bibliography 16

1. Assignment Description

During the summer course HPCP 2025 several methods to enhance performance of Python code were discussed and tested. The focus was set on improvements based in GPU manipulation including:

- CPU vectorization (NumPy, Numba, exploiting CPU vector units)
- GPU acceleration (CuPy, Numba CUDA)
- MPI (multi-process, multi-GPU)
- Dask (distributed arrays and dataframes on multi-GPU)
- Hybrid approaches (Marcin, 2025)

This report is based on the assignment for the end of the HPCP course. The Assignment is based on the DEMREG codebase.

The input to DEMREG consists of solar images from NASA's Atmospheric Imaging Assembly (AIA) camera on board the Solar Dynamics Observatory (SDO). This camera captures a full-disc image of the sun every 12 seconds in ten different wavelengths which give insight into the processes inside the solar atmosphere and the solar surface.

For this project, six of those ten wavelengths are used because they give the best insight into the temperature of the sun. For each pixel across these 6 bands, the code reconstructs the temperature distribution of the solar plasma. The result is a temperature map of the Sun derived from overlapping pixel intensities.(Marcin, 2025)

The Goal of the Assignment is to improve the code base performance in 4 different ways based on measurements done by time the execution time.

1.1. General approach

While studying the demreg codebase we discovered that there are two main jupyter notebooks.

- example_demregpy_aiasyn.ipynb
- example_demregpy_aiapxl.ipynb

In an initial experimentation phase, we downloaded .fits files using a snippet provided in a notebook and ran the code of the example_demregpy_aiapxl.ipynb notebook. But for simplicity and consistency we decided to work with the example_demregpy_aiasyn.ipynb notebook which uses synthetic data for its calculation. More specifically, we took a deeper look at the main demreg calculation, the "dn2dem_pos" function, found in the "dn2dem_pos.py" file.

2. Workflow

For the code optimization we took the following approach:

1. Run the “de2dem_pos” function and profile it with “cProfile” and measure the time with “%timeit”.
2. Display the profiler data as an icicle graph with “snakeviz” and analyze the different parts of the function.
3. Find bottlenecks in the code and find approaches to optimize them.
4. After implementing an approach to enhance the performance, we ran the profiler and “%timeit” again to see whether our changes make a difference.
5. Compare the different approaches.

2.1. Step 1: Profiling the “de2dem_pos” function

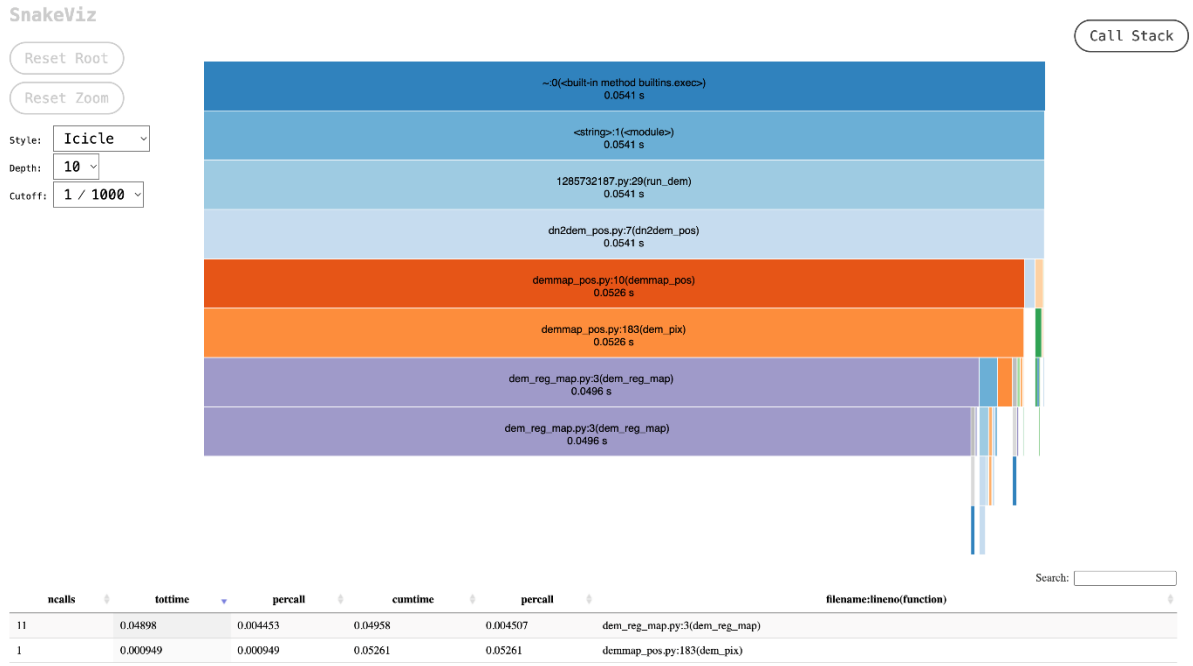
1. Profile and time the “de2dem_pos”.

```
import cProfile

def run_dem():
    return dn2dem_pos(dn_in5, edn_in5, trmatrix5, tresp_logt, temps)

# Warm up
for _ in range(3):
    _ = run_dem()

# Time it
%timeit run_dem()

# Run profiler
cProfile.run('run_dem()', 'profile_output_syn.prof')
```

2. Analyze the results with “snakeviz”

ncalls	tottime	percall	cumtime	percall	
11	0.06919	0.00629	0.07015	0.006377	dem_reg_map.py:3(dem_reg_map)
4	0.001533	0.0003832	0.00165	0.0004126	linalg.py:1499(svd)
108	0.0009343	8.651e-06	0.0009343	8.651e-06	~:0(<method 'reduce' of 'numpy.ufunc' ob

3. The profiling revealed that “dem_reg_map” takes the most time. We checked the total time in the table in snakeviz.



4. Time the function

```
53.3 ms ± 266 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

3. Optimizations

After the profiling, one part stood out the most. The “dem_reg_map” function took the most time. This function executes a set of element-wise operations on vectors/scalars which scale with the size of the matrix/vectors provided and use a lot of execution time.

```
step=(np.log(maxx)-np.log(minx))/(nmu-1.)
mu=np.exp(np.arange(nmu)*step)*minx
for kk in np.arange(nf):
    coef=data@U[kk,:]
    for ii in np.arange(nmu):
        arg[kk,ii]=(mu[ii]*sigmab[kk]**2*coef/(sigmaa[kk]**2+mu[ii]*sigmab[kk]*
**2))**2
```

For the following optimizations, we will focus on this part of the code.

3.1. "dem_reg_map" with NumPy Vectorization

The first approach was NumPy Vectorization. The code is on the branch “feature/numpy-vectorization”.

```python
def dem_reg_map_vectorized(sigmaa, sigmab, U, W, data, err, reg_tweak,
nmu=500):
    nf = data.shape[0]

    sigs = sigmaa[:nf] / sigmab[:nf]

    maxx = np.max(sigs)
    minx = np.min(sigs) ** 2.0 * 1E-4
    step = (np.log(maxx) - np.log(minx)) / (nmu - 1.)
    mu = np.exp(np.arange(nmu) * step) * minx

    # Matrix multiplication
    coef = data @ U[:nf, :].T

    mu_2d = mu[np.newaxis, :]
    sigmaa_2d = sigmaa[:nf, np.newaxis]
    sigmab_2d = sigmab[:nf, np.newaxis]
    coef_2d = coef[:, np.newaxis]

    numerator = mu_2d * sigmab_2d ** 2 * coef_2d
    denominator = sigmaa_2d ** 2 + mu_2d * sigmab_2d ** 2
    arg = (numerator / denominator) ** 2

    discr = np.sum(arg, axis=0) - np.sum(err ** 2) * reg_tweak

    opt = mu[np.argmin(np.abs(discr))]

    return opt
```

The original code used two nested Python loops. The vectorized version uses NumPy arrays to perform operations on entire arrays at once.

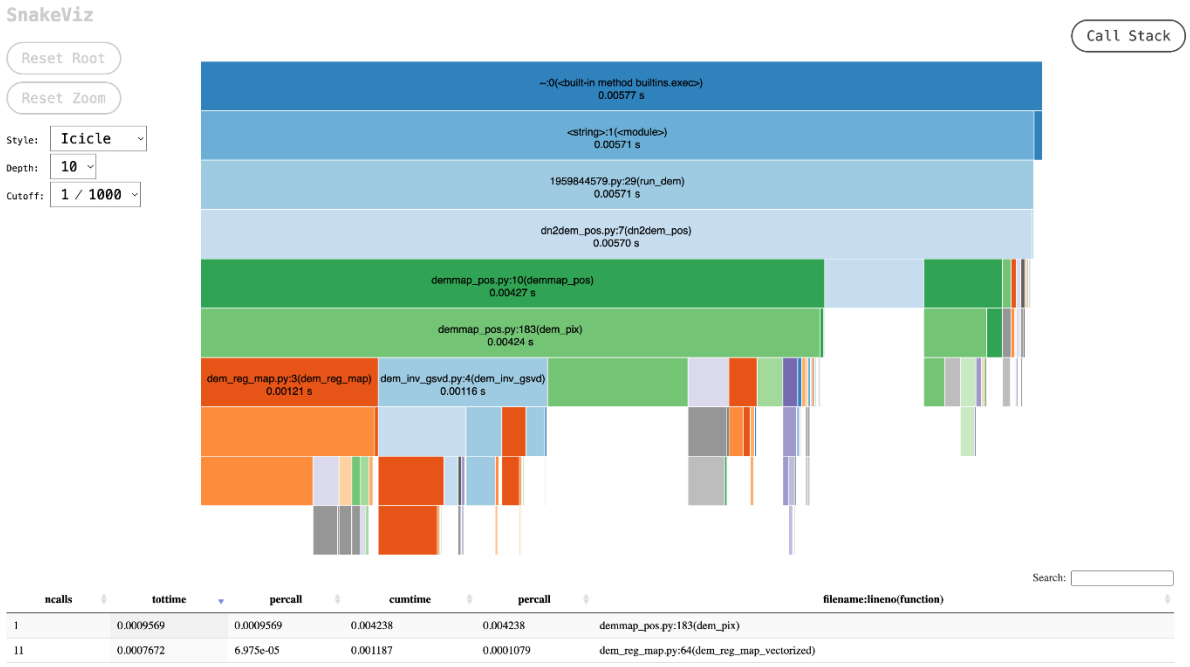
3.1.1. Performance boost

After the application of the NymPy Vectorization the total time went from 0.04898 seconds to 0.00438 seconds. %timeit spat out this result:

`4.38 ms ± 23 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)`

(previous) `53.3 ms ± 266 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)`

3.1.2. Profiling after NymPy Vectorization

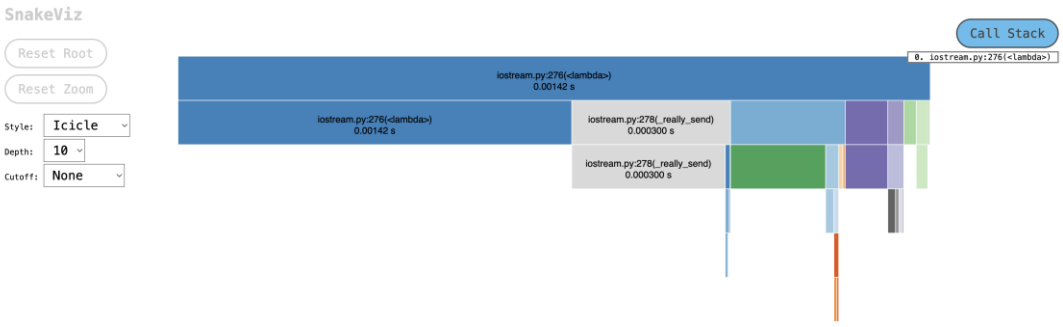


ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1	0.0009569	0.0009569	0.004238	0.004238	demmap_pos.py:183(dem_pix)
11	0.0007672	6.975e-05	0.001187	0.0001079	dem_reg_map.py:64(dem_reg_map_vectorized)

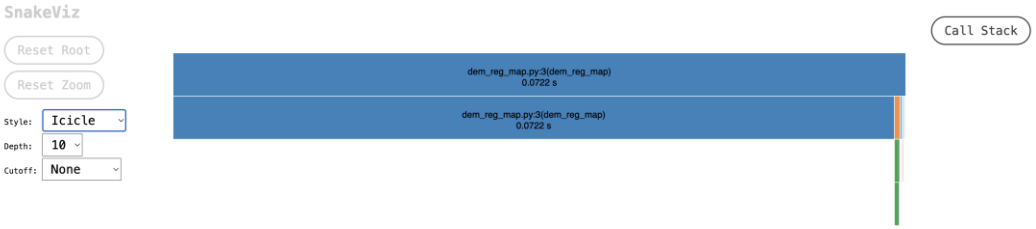
3.1.3. Confusion and warm up

During the first few profiling attempts we got really confused because seemingly every time we profiled the same code, we got a different looking graph. Sometimes the actual methods were completely hidden and sometimes not.

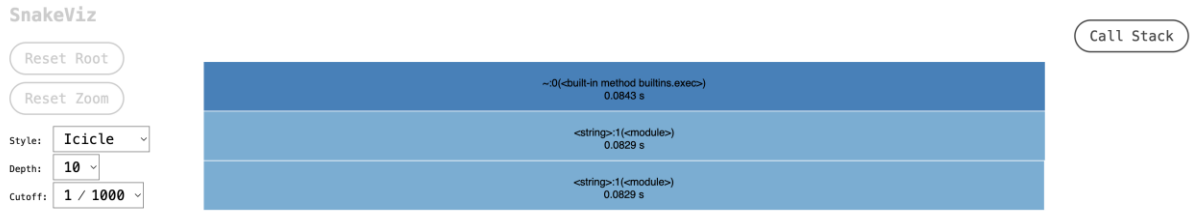
Result 1:



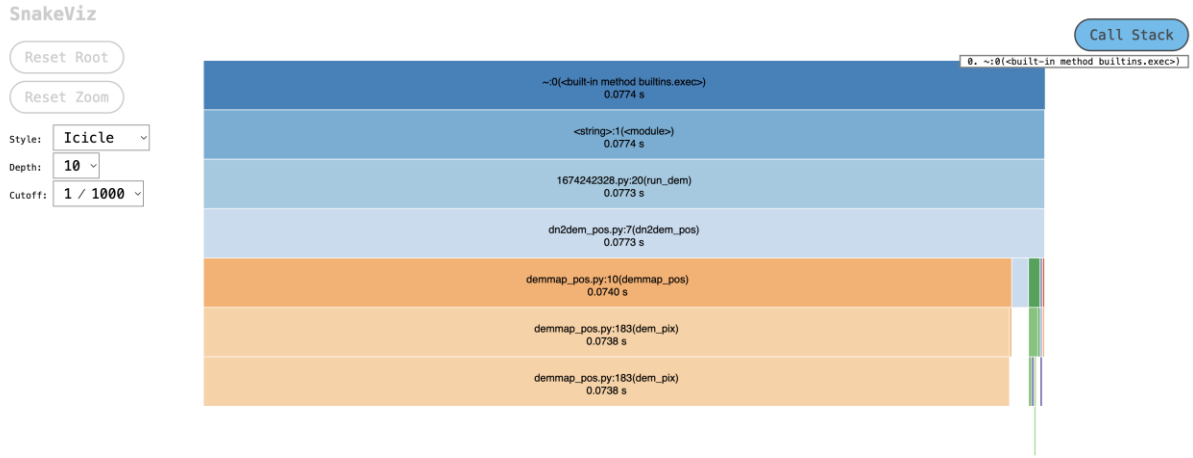
Result2:



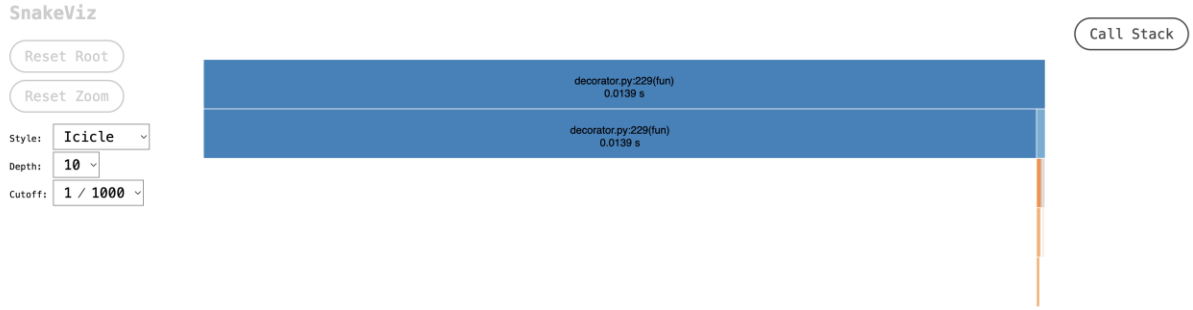
Result 3:



Result 4:



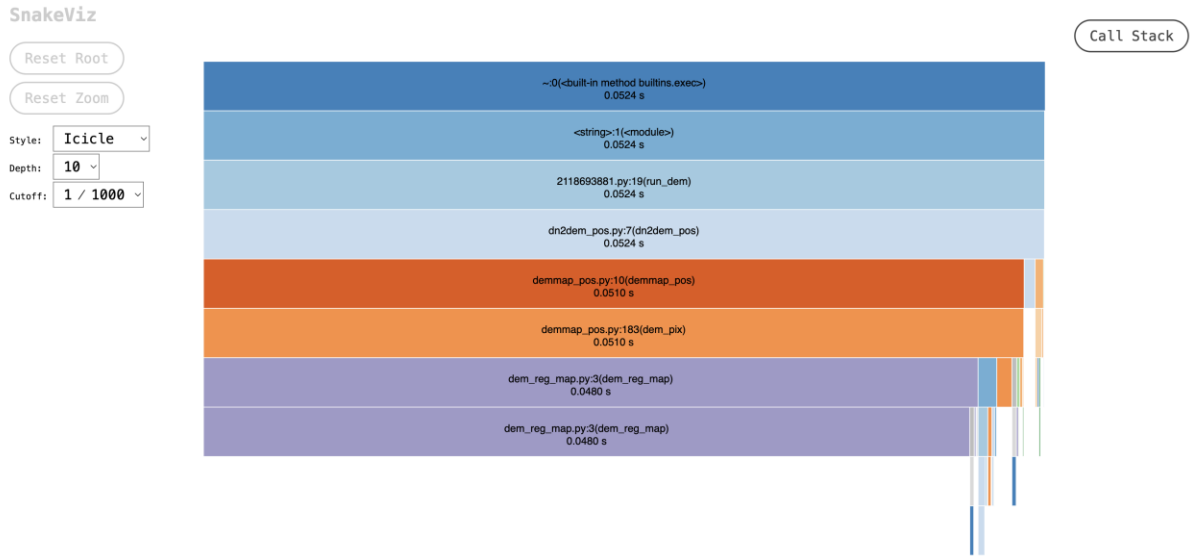
Result 5:



Then we called the function a few times before profiling (warm up). We got consistent results after doing that. We remembered that this was a topic we discussed in class and concluded this phenomenon was due to setup and caching of NumPy.

```
def run_dem():
    return dn2dem_pos(dn_in5, edn_in5, trmatrix5, tresp_logt, temps)

for _ in range(3):  # warm up runs
    _ = run_dem()

# Run profiler
cProfile.run('run_dem()', 'profile_output.prof')
```

The final profiling looked like this:



3.2. "dem_reg_map" with Numba / JIT

The second approach was done with Numba / Jit. The code is on the branch "features/numba_jit".

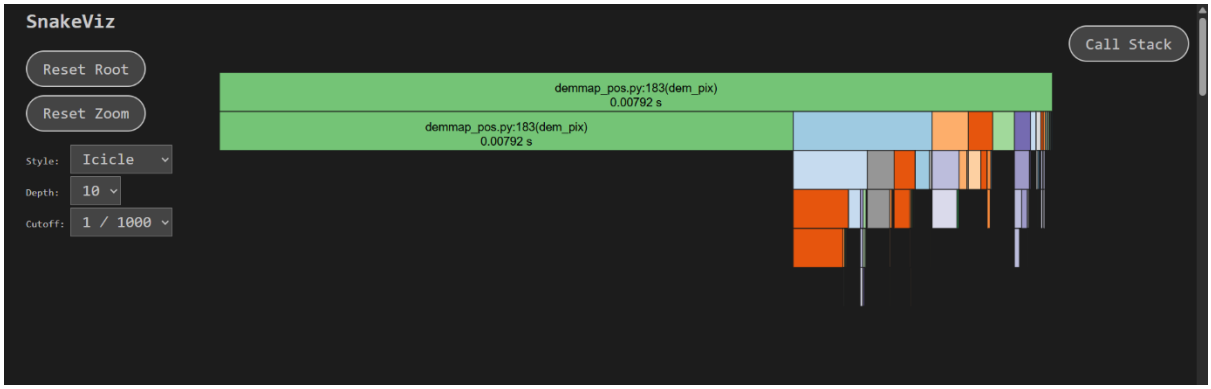
```python
import numpy as np
from numba import njit, prange

@njit(parallel=True, fastmath=True)
def dem_reg_map(sigmaa, sigmab, U, W, data, err, reg_tweak, nmu=500):
    nf = data.shape[0]
    nreg = sigmaa.shape[0]

    arg = np.zeros((nreg, nmu))
    discr = np.zeros(nmu)

    sigs = sigmaa[:nf] / sigmab[:nf]
    maxx = np.max(sigs)
    minx = np.min(sigs) ** 2.0 * 1e-4

    step = (np.log(maxx) - np.log(minx)) / (nmu - 1.0)
    mu = np.exp(np.arange(nmu) * step) * minx

    for kk in prange(nf):
        coef = 0.0
        # Equivalent to coef = data @ U[kk, :]
        for jj in range(U.shape[1]):
            coef += data[jj] * U[kk, jj]

        for ii in range(nmu):
            denom = sigmaa[kk] ** 2 + mu[ii] * sigmab[kk] ** 2
            val = mu[ii] * sigmab[kk] ** 2 * coef / denom
            arg[kk, ii] = val * val

    for ii in prange(nmu):
        s = 0.0
        for kk in range(nreg):
            s += arg[kk, ii]
        discr[ii] = s - np.sum(err ** 2) * reg_tweak

    opt = mu[np.argmin(np.abs(discr))]
    return opt
```

3.2.1. Performance boost

`7.69 ms ± 174 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)`

(previous) `53.3 ms ± 266 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)`

3.2.2. Profiling after Numba / Jit



3.2.3. Conclusion

With the flag `parallel` enabled, the for-loops are parallelized. The `fastmath` flag exploits the ability of Numba to vectorize the actions in the for loop because it does not need to be executed in a certain order. The `data @ U[kk, :]` has been replaced with the manual dot product and inside the loops only arithmetic operations are used because NumPy calls cannot be optimized by Numba.

3.3. "dem_reg_map" with CuPy

The fourth approach was done with CuPy. The code is on the branch “cupy”.

Firstly, we simplified the operations that are happening in dem_reg_map by removing a loop and then sending most of the input data to the GPU and making the main Calculation on the GPU.

```
import numpy as np
import cupy as cp

def dem_reg_map(sigmaa, sigmab, U, W, data, err, reg_tweak, nmu=500):
    nf = data.shape[0]
    nreg = sigmaa.shape[0]

    sigs = sigmaa[:nf] / sigmab[:nf]
    maxx = max(sigs)
    minx = min(sigs) ** 2.0 * 1E-4

    step = (np.log(maxx) - np.log(minx)) / (nmu - 1.)
    mu = np.exp(np.arange(nmu) * step) * minx

    data_gpu = cp.asarray(data, dtype=cp.float64)
    U_gpu = cp.asarray(U, dtype=cp.float64)
    sigmaa_gpu = cp.asarray(sigmaa, dtype=cp.float64)
    sigmab_gpu = cp.asarray(sigmab, dtype=cp.float64)
    mu_gpu = cp.asarray(mu, dtype=cp.float64)

    nf = U_gpu.shape[0]
    nmu = mu_gpu.shape[0]

    arg_gpu = cp.empty((nf, nmu))  # allocate result

    coef_gpu = data_gpu @ U_gpu.T
    sa2_gpu = sigmaa_gpu ** 2
    sb2_gpu = sigmab_gpu ** 2

    top = (sb2_gpu[:, None] * coef_gpu[:, None]) * mu_gpu[None, :]
    bot = sa2_gpu[:, None] + sb2_gpu[:, None] * mu_gpu[None, :]
    arg_gpu = (top / bot) ** 2

    discr = cp.sum(arg_gpu, axis=0).get() - np.sum(err ** 2) * reg_tweak

    opt = mu[np.argmin(np.abs(discr))]

    return opt
```

3.3.1. Result

We could not observe a performance boost. One Problem was: Inside the demreg_aiasyn notebook, there would be calculation started with multiprocessing, which led to our changed code being executed inside a Subprocess, which led to a crash of cupy. This part here:

```
# do we have enough DEM's to make parallel make sense?
if (na >= 200):
    n_par = 100
    niter = int(np.floor((na) / n_par))
    with threadpool_limits(limits=1):
```

We tried to run it using the else branch there (serial), but that was quite slow (not surprising).

We found without changing this condition, we could prevent the CuPy crash like this:

```
import multiprocessing as mp
mp.set_start_method("spawn", force=True)
```

But we suppose the overhead for creating a process like this for every calculation is way too big.

We don’t really understand how to get this setup properly.

3.4. "dem_reg_map" with Numba / JIT (no parallel)

The fourth approach was done similarly to the second approach, with “Numba / Jit” but with different settings. This time the flags “fastmath” and “parallel” were not set to true. This again improved the performance. This code is on the branch ”features/numba_jit_no_flags”

```
import numpy as np
from numba import njit

@njit  # No parallel or fastmath
def dem_reg_map(sigmaa, sigmab, U, W, data, err, reg_tweak, nmu=500):

    nf = data.shape[0]
    nreg = sigmaa.shape[0]

    arg = np.zeros((nreg, nmu))
    discr = np.zeros(nmu)

    sigs = sigmaa[:nf] / sigmab[:nf]
    maxx = np.max(sigs)
    minx = np.min(sigs)**2.0 * 1e-4

    step = (np.log(maxx) - np.log(minx)) / (nmu - 1.0)
    mu = np.exp(np.arange(nmu) * step) * minx

    for kk in range(nf):
        coef = 0.0
        for jj in range(U.shape[1]):
            coef += data[jj] * U[kk, jj]
        for ii in range(nmu):
            val = mu[ii] * sigmab[kk]**2 * coef / (sigmaa[kk]**2 + mu[ii] *
sigmab[kk]**2)
            arg[kk, ii] = val * val

    # Sum over first axis manually (Numba prefers explicit loops)
    for ii in range(nmu):
        s = 0.0
        for kk in range(nreg):
            s += arg[kk, ii]
        discr[ii] = s - np.sum(err**2) * reg_tweak
```

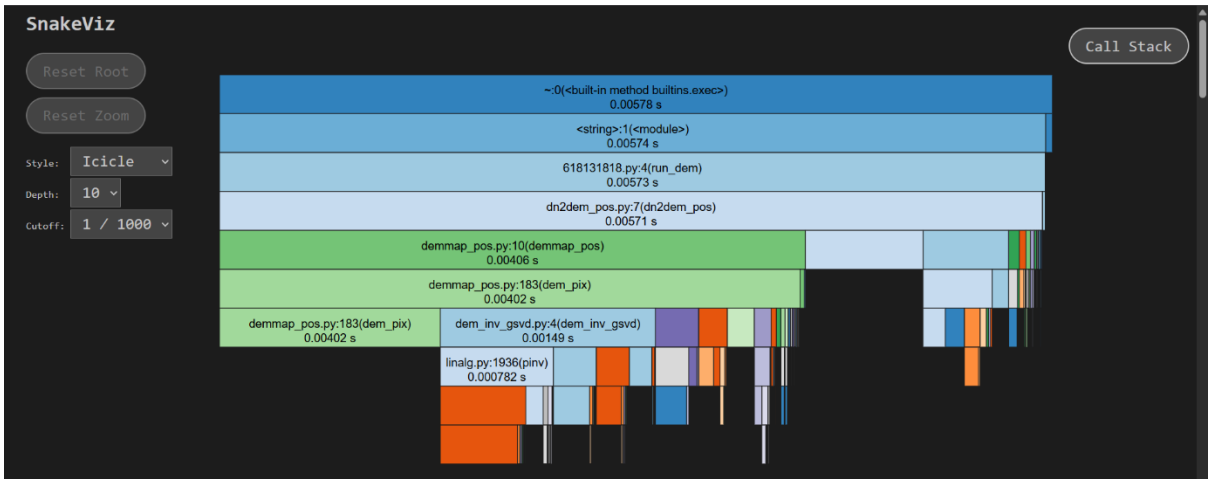
```
    opt_index = 0
    min_abs_discr = abs(discr[0])
    for ii in range(1, nmu):
        abs_val = abs(discr[ii])
        if abs_val < min_abs_discr:
            min_abs_discr = abs_val
            opt_index = ii

    return mu[opt_index]
```

3.4.1. Performance boost

`4.1 ms ± 37.1 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)`

(previous) `53.3 ms ± 266 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)`

3.4.2. Profiling after



3.4.3. Conclusion

The matrix multiplication and other functions like np.sum have been replaced with manual loops / calculations because Numba is good at optimizing these operations instead of numpy functions. `arange` got replaced with `range` so no temporary array is created and less memory / calculations are needed.

Interestingly, comparing this version to the other version (see chapter 3.2), running without parallelization and fastmath was actually more performant. When using `parallel=True`, there is an overhead when scheduling the threads. It might be better when using the real data but using the synthetic data the overhead outweighs the performance gains. The `fastmath` flag can be faster in general but modern CPU already

have very optimized floating-point operations. In general, these calculations are probably memory-bound, so the flag optimizations are unnecessary in our case.

3.5. Overall conclusion

In our experiments we found out that using NumPy and JIT (no parallel) was the fastest with 4.1ms compared to 53.3ms in the original run. This is 13 times faster! We were using synthetic data so it is possible that using real data other approaches will be faster.

Bibliography

Marcin, S. (2025). *Assignment READ.ME*.