GitHub: https://github.com/DayBeha/HPC_assignment

Exercise 1 - Cythonize the STREAM Benchmark

Task 1.1: Develop a Cython version of the STREAM benchmark. Make sure to define the *ctypes* for obtaining full performance.

Cython Code as follows:

```
#cython: language_level=3

import numpy as np
cimport numpy as np

def copy(float[:] a, float[:] b):
    # cdef float[:] c = np.empty(len(a), dtype=np.float32)
    for i in range(len(a)):
        b[i] = a[i]
    # return c

def scale(float scalar, float[:] a, float[:] b):
    # cdef float[:] b = np.empty(len(a), dtype=np.float32)
    for i in range(len(a)):
        b[i] = scalar * a[i]
    # return b

def add(float[:] a, float[:] b, float[:] c):
    # cdef float[:] c = np.empty(len(a), dtype=np.float32)
    for i in range(len(a)):
        c[i] = a[i] + b[i]
    # return c

def triad(float[:] a, float scalar, float[:] b, float[:] c):
    # cdef float[:] c = np.empty(len(a), dtype=np.float32)
    for i in range(len(a)):
        c[i] = a[i] + scalar * b[i]
    # return c
```

Task 1.2: Plot the bandwidth results varying the arrays' size. Answer the question: how does the bandwidth measured with Cython code compare to bandwidth obtained in Assignment II.

bandwidth before Cython:

STREAM_ARRAY_SIZE = 5

Copy	time use:8.00 us	data amount:280	bandwith:35.000 MB/s
add	time use:87.00 us	data amount:280	bandwith:3.218 MB/s
scale	time use:152.00 us	data amount:420	bandwith:2.763 MB/s

triad time use:31.00 us data amount:420 bandwith:13.548 MB/s

STREAM_ARRAY_SIZE = 10
Copy time use:2.00 us data amount:560 bandwith:280.000 MB/s
add time use:10.00 us data amount:560 bandwith:56.000 MB/s
scale time use:21.00 us data amount:840 bandwith:40.000 MB/s
triad time use:18.00 us data amount:840 bandwith:46.667 MB/s

STREAM_ARRAY_SIZE = 50
Copy time use:1.00 us data amount:2800 bandwith:2800.000 MB/s
add time use:8.00 us data amount:2800 bandwith:350.000 MB/s
scale time use:16.00 us data amount:4200 bandwith:262.500 MB/s
triad time use:18.00 us data amount:4200 bandwith:233.333 MB/s

STREAM_ARRAY_SIZE = 100
Copy time use:2.00 us data amount:5600 bandwith:2800.000 MB/s
add time use:8.00 us data amount:5600 bandwith:700.000 MB/s
scale time use:18.00 us data amount:8400 bandwith:466.667 MB/s
triad time use:21.00 us data amount:8400 bandwith:400.000 MB/s

STREAM_ARRAY_SIZE = 500
Copy time use:4.00 us data amount:28000 bandwith:7000.000 MB/s
add time use:10.00 us data amount:28000 bandwith:2800.000 MB/s
scale time use:22.00 us data amount:42000 bandwith:1909.091 MB/s
triad time use:25.00 us data amount:42000 bandwith:1680.000 MB/s

STREAM_ARRAY_SIZE = 1000
Copy time use:4.00 us data amount:56000 bandwith:14000.000 MB/s
add time use:12.00 us data amount:56000 bandwith:4666.667 MB/s
scale time use:21.00 us data amount:84000 bandwith:4000.000 MB/s
triad time use:25.00 us data amount:84000 bandwith:3360.000 MB/s

STREAM_ARRAY_SIZE = 5000
Copy time use:4.00 us data amount:280000 bandwith:70000.000 MB/s
add time use:17.00 us data amount:280000 bandwith:16470.588 MB/s
scale time use:26.00 us data amount:420000 bandwith:16153.846 MB/s
triad time use:91.00 us data amount:420000 bandwith:4615.385 MB/s

STREAM_ARRAY_SIZE = 10000
Copy time use:4.00 us data amount:560000 bandwith:140000.000 MB/s
add time use:36.00 us data amount:560000 bandwith:15555.556 MB/s
scale time use:52.00 us data amount:840000 bandwith:16153.846 MB/s
triad time use:60.00 us data amount:840000 bandwith:14000.000 MB/s

STREAM_ARRAY_SIZE = 50000

Copy time use:18.00 us data amount:2800000 bandwith:155555.556 MB/s

add time use:347.00 us data amount:2800000 bandwith:8069.164 MB/s

scale time use:242.00 us data amount:4200000 bandwith:17355.372 MB/s

triad time use:302.00 us data amount:4200000 bandwith:13907.285 MB/s

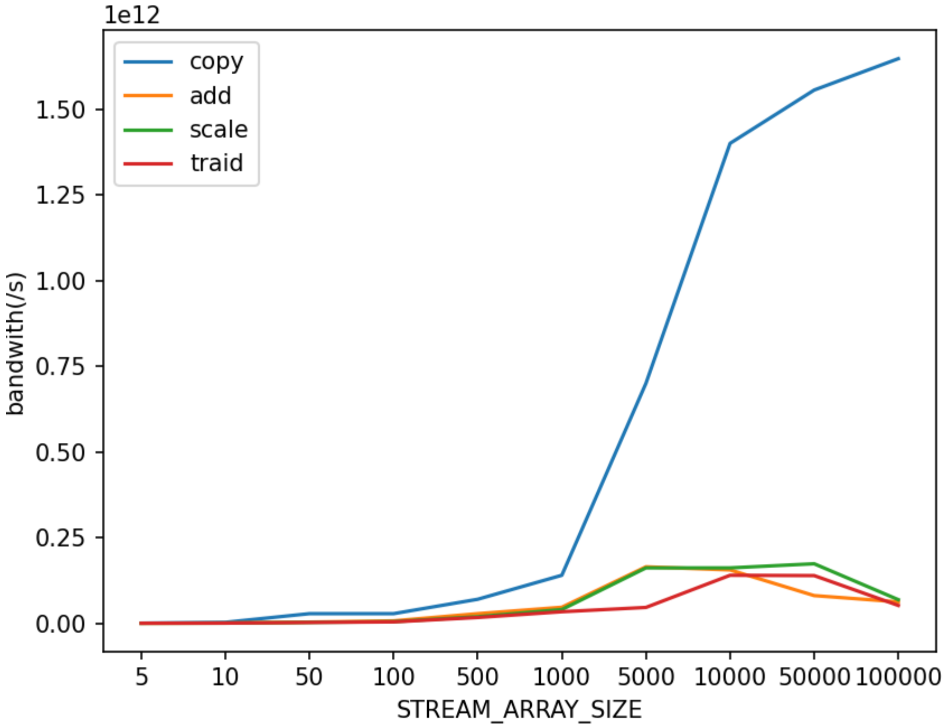
STREAM_ARRAY_SIZE = 100000

Copy time use:34.00 us data amount:5600000 bandwith:164705.882 MB/s

add time use:898.00 us data amount:5600000 bandwith:6236.080 MB/s

scale time use:1220.00 us data amount:8400000 bandwith:6885.246 MB/s

triad time use:1616.00 us data amount:8400000 bandwith:5198.020 MB/s



bandwidth after Cython:

STREAM_ARRAY_SIZE = 5

Copy time use:83.00 us data amount:280 bandwith:3.373 MB/s

add time use:29.00 us data amount:280 bandwith:9.655 MB/s

scale time use:10.00 us data amount:420 bandwith:42.000 MB/s

triad time use:11.00 us data amount:420 bandwith:38.182 MB/s

STREAM_ARRAY_SIZE = 10

Copy time use:13.00 us data amount:560 bandwith:43.077 MB/s

add time use:11.00 us data amount:560 bandwith:50.909 MB/s

scale time use:8.00 us data amount:840 bandwith:105.000 MB/s

triad time use:9.00 us data amount:840 bandwith:93.333 MB/s

STREAM_ARRAY_SIZE = 50

Copy time use:9.00 us data amount:2800 bandwith:311.111 MB/s
add time use:9.00 us data amount:2800 bandwith:311.111 MB/s
scale time use:7.00 us data amount:4200 bandwith:600.000 MB/s
triad time use:9.00 us data amount:4200 bandwith:466.667 MB/s

STREAM_ARRAY_SIZE = 100
Copy time use:9.00 us data amount:5600 bandwith:622.222 MB/s
add time use:10.00 us data amount:5600 bandwith:560.000 MB/s
scale time use:7.00 us data amount:8400 bandwith:1200.000 MB/s
triad time use:10.00 us data amount:8400 bandwith:840.000 MB/s

STREAM_ARRAY_SIZE = 500
Copy time use:12.00 us data amount:28000 bandwith:2333.333 MB/s
add time use:15.00 us data amount:28000 bandwith:1866.667 MB/s
scale time use:10.00 us data amount:42000 bandwith:4200.000 MB/s
triad time use:15.00 us data amount:42000 bandwith:2800.000 MB/s

STREAM_ARRAY_SIZE = 1000
Copy time use:17.00 us data amount:56000 bandwith:3294.118 MB/s
add time use:22.00 us data amount:56000 bandwith:2545.455 MB/s
scale time use:14.00 us data amount:84000 bandwith:6000.000 MB/s
triad time use:22.00 us data amount:84000 bandwith:3818.182 MB/s

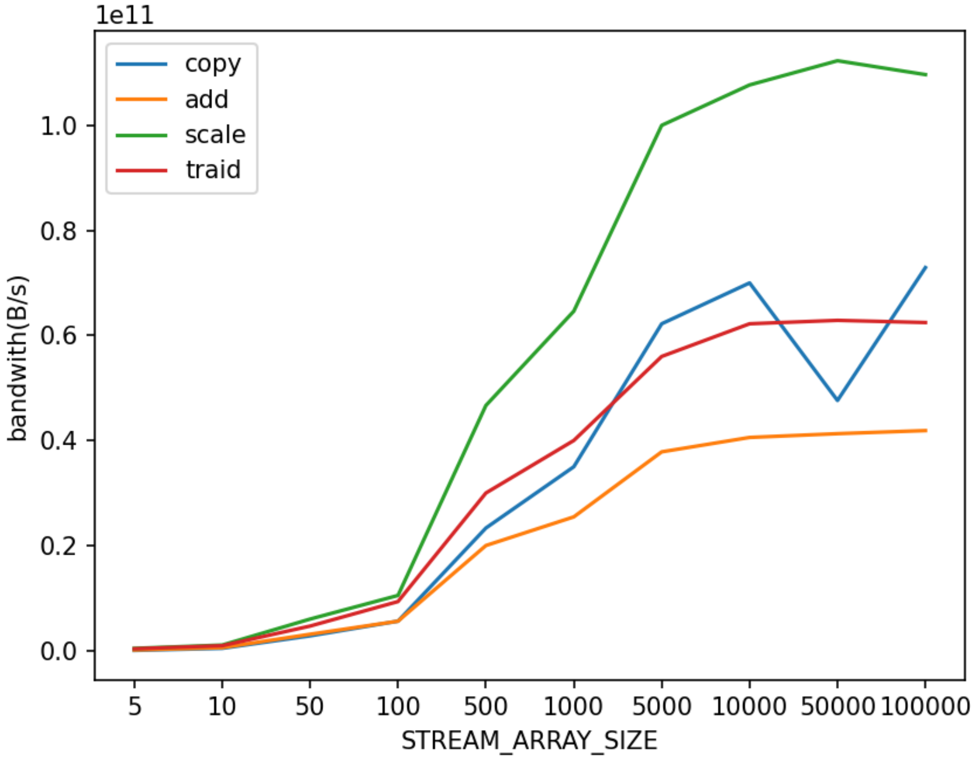
STREAM_ARRAY_SIZE = 5000
Copy time use:45.00 us data amount:280000 bandwith:6222.222 MB/s
add time use:74.00 us data amount:280000 bandwith:3783.784 MB/s
scale time use:43.00 us data amount:420000 bandwith:9767.442 MB/s
triad time use:74.00 us data amount:420000 bandwith:5675.676 MB/s

STREAM_ARRAY_SIZE = 10000
Copy time use:82.00 us data amount:560000 bandwith:6829.268 MB/s
add time use:137.00 us data amount:560000 bandwith:4087.591 MB/s
scale time use:79.00 us data amount:840000 bandwith:10632.911 MB/s
triad time use:141.00 us data amount:840000 bandwith:5957.447 MB/s

STREAM_ARRAY_SIZE = 50000
Copy time use:385.00 us data amount:2800000 bandwith:7272.727 MB/s
add time use:672.00 us data amount:2800000 bandwith:4166.667 MB/s
scale time use:372.00 us data amount:4200000 bandwith:11290.323 MB/s
triad time use:681.00 us data amount:4200000 bandwith:6167.401 MB/s

STREAM_ARRAY_SIZE = 100000
Copy time use:749.00 us data amount:5600000 bandwith:7476.636 MB/s
add time use:1340.00 us data amount:5600000 bandwith:4179.104 MB/s

scale time use:741.00 us data amount:8400000 bandwith:11336.032 MB/s
triad time use:1355.00 us data amount:8400000 bandwith:6199.262 MB/s



Conclusion:

1、As the plots shows, ‘copy’ get the highest bandwidth and high better than others before Cythonizing and ‘scale’ get the highest bandwidth after Cythonizing. It may because it’s copied as matrix before Cythonizing.

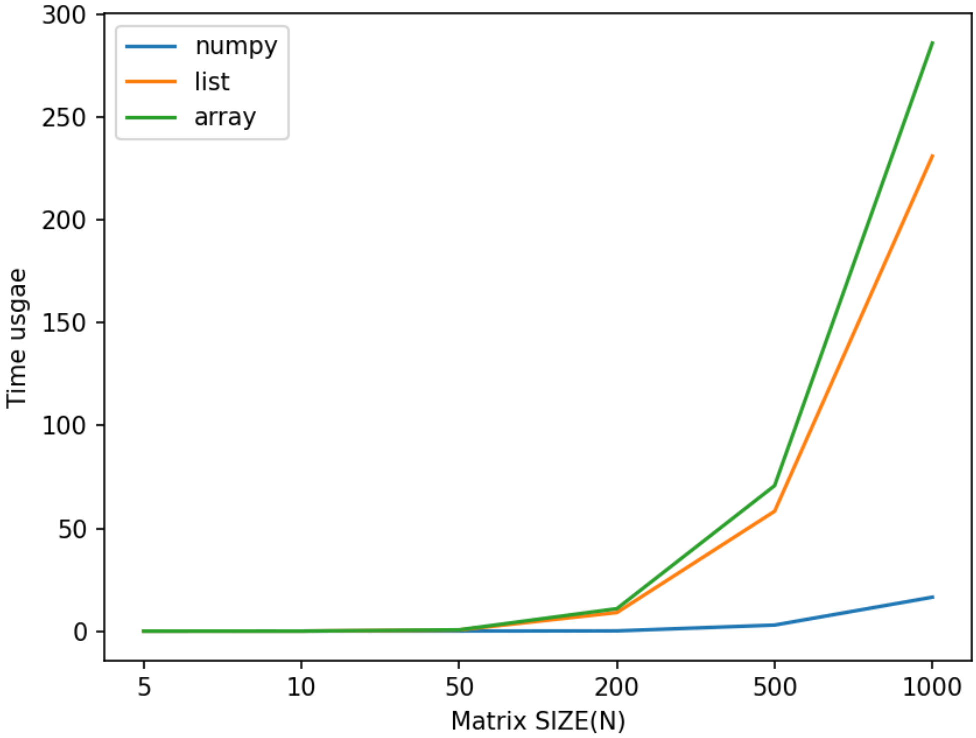
2、The performance after Cythonizing is no better than before which not as expectation. I both tried pass variable to functions in *.pyx and define a new array. No better performance appear.

3、The bandwidth decrease when the STREAM_ARRAY_SIZE is too large(100000) before Cythonizing but not decrease after Cythonizing where bandwidth after Cythonizing is better than before. It may because ‘np.array’ need more memory than ‘cdef’.

Exercise 2 - Gauss-Seidel for Poisson Solver

Task 2.1: Develop the Gauss-Seidel solver with Python List, array, or NumPy. Plot the performance varying the grid size.

Set grid size as [5,10,50,200,500,1000]




```
Numpy time   average:32650.33 ms    min:317.35 ms   max:154148.00 ms
 list time   average:515427.62 ms   min:28.35 ms    max:2418517.64 ms
 array time  average:628008.64 ms   min:33.97 ms    max:2949629.43 ms

------------FLOPS/s-------------
```

	numpy	list	array
5	7521.692561346932	88149.21899791942	73570.52470498308
10	63017.13435883211	122703.90321116058	99343.34051916863
50	6225316.619603266	470853.1538968467	389149.39414109127
200	134905431.29266375	1799314.2700859562	1471214.8944104011
500	63264281.550953686	4313136.6837794045	3556140.889391718
1000	129745408.30513468	8269528.148778352	6780512.704508332

The performance is Numpy>list>array.

Task 2.2: Profile the code to identify the part of the code to optimize. You can use the tool of your choice.

line_profiler

```
Timer unit: 1e-06 s

Total time: 2.7237 s
File: .\e2GaussSeidel.py
Function: gauss_seidel at line 13

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    13                                           @profile
    14                                           def gauss_seidel(f):
    15      1000        718.3      0.7      0.0      newf = f.copy()
    16     48000       8986.6      0.2      0.3      for i in range(1, len(newf) - 1):
    17   2304000     440935.2      0.2     16.2          for j in range(1, len(newf[0]) - 1):
    18   4608000    1226510.6      0.3     45.0              newf[i][j] = 0.25 * (newf[i][j + 1] + newf[i][j - 1] +
    19   4608000    1046318.9      0.2     38.4                                   newf[i + 1][j] + newf[i - 1][j])
    20      1000        228.0      0.2      0.0      return newf
```

memory_profiler

```
Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
    13    81.254 MiB    31.961 MiB        1000   @profile
    14                                          def gauss_seidel(f):
    15    81.254 MiB   -49.254 MiB        1000       newf = f.copy()
    16    81.254 MiB -2415.875 MiB       49000        for i in range(1, len(newf) - 1):
    17    81.254 MiB -115960.996 MiB     2352000            for j in range(1, len(newf[0]) - 1):
    18    81.254 MiB -340783.230 MiB     6912000                newf[i][j] = 0.25 * (newf[i][j + 1] + newf[i][j - 1] +
    19    81.254 MiB -227188.820 MiB     4608000                                     newf[i + 1][j] + newf[i - 1][j])
    20    81.254 MiB   -49.340 MiB        1000       return newf
```

It can be seen that the code in loop takes the most time.

Task 2.3: Use the Cython Annotation tool to identify the parts to use Cython

```
+06: def gauss_seidel(f):
+07:     newf = f.copy()
 08:
+09:     for i in range(1, newf.shape[0] - 1):
+10:         for j in range(1, newf.shape[1] - 1):
+11:             newf[i, j] = 0.25 * (newf[i, j + 1] + newf[i, j - 1] +
+12:                                  newf[i + 1, j] + newf[i - 1, j])
 13:
+14:     return newf
```

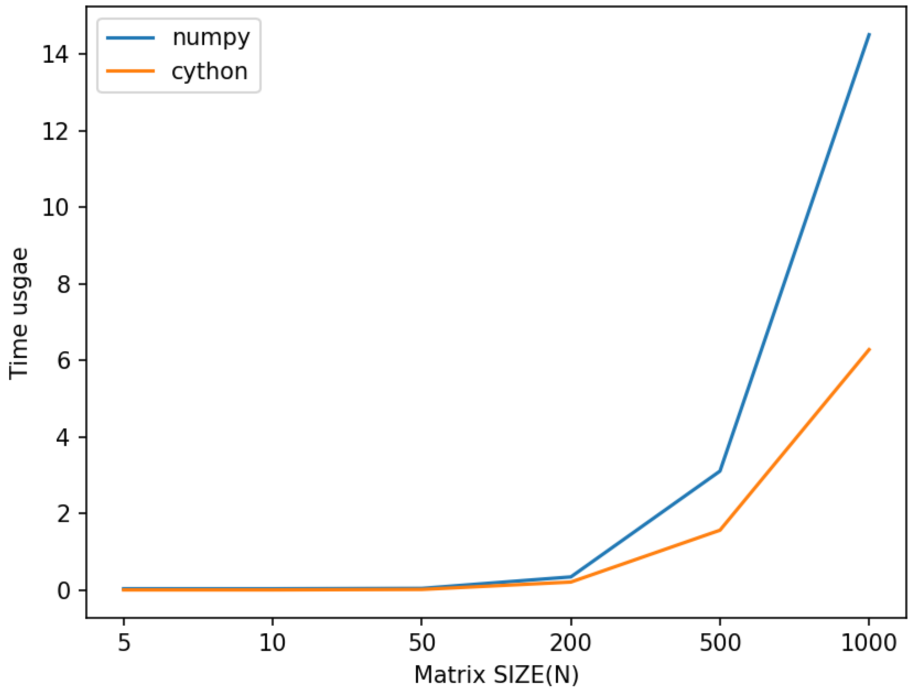
It's too yellow here, so all the code need to be update.

Task 2.4: Use Cython to optimize the part you identified as the most computationally expensive. Compare the performance with the results obtained in Task 2.1.

After optimize:

```
#cython: language_level=3

def gauss_seidel(float[:, :] f, int N):
    cdef float[:, :] newf = f.copy()
     cdef int i, j    # warning if not declare here: gauss_seidel.p
    for i in range(1, N - 1):
        for j in range(1, N - 1):
            newf[i, j] = 0.25 * (newf[i, j + 1] + newf[i, j - 1] +
                                 newf[i + 1, j] + newf[i - 1, j])

    return newf
```

Compare performance:




```
Numpy time   average:30091.61 ms     min:316.78 ms    max:145046.73 ms
Cython time  average:13433.94 ms     min:8.53 ms      max:62791.82 ms

------------FLOPS/s-------------
+------+-------------------+-------------------+
|      |       numpy       |       Cython      |
+------+-------------------+-------------------+
|  5   | 7795.301615810118 | 292740.0468384001 |
|  10  | 63130.71530256975 | 1900057.0017099248 |
|  50  | 6112095.837662742 | 19530639.66751049 |
| 200  | 46955121.76196747 | 77659196.49912347 |
| 500  | 80517813.27790272 | 160209446.29500592 |
| 1000 | 137886552.28099278 |  318512581.915862  |
+------+-------------------+-------------------+
```

It looks better after cython is applied, taking about half time compared with numpy.

Task 2.5: Use PyTorch to port your code to Nvidia GPUs. For this you will need to express the two nested loops operations as numpy roll operations in 2D as we did for the diffusion code.

Task 2.6: Use CuPy to port your code to Nvidia GPUs. See [C.3 Tutorial - Introduction to CuPyC.3 Tutorial - Introduction to CuPy]

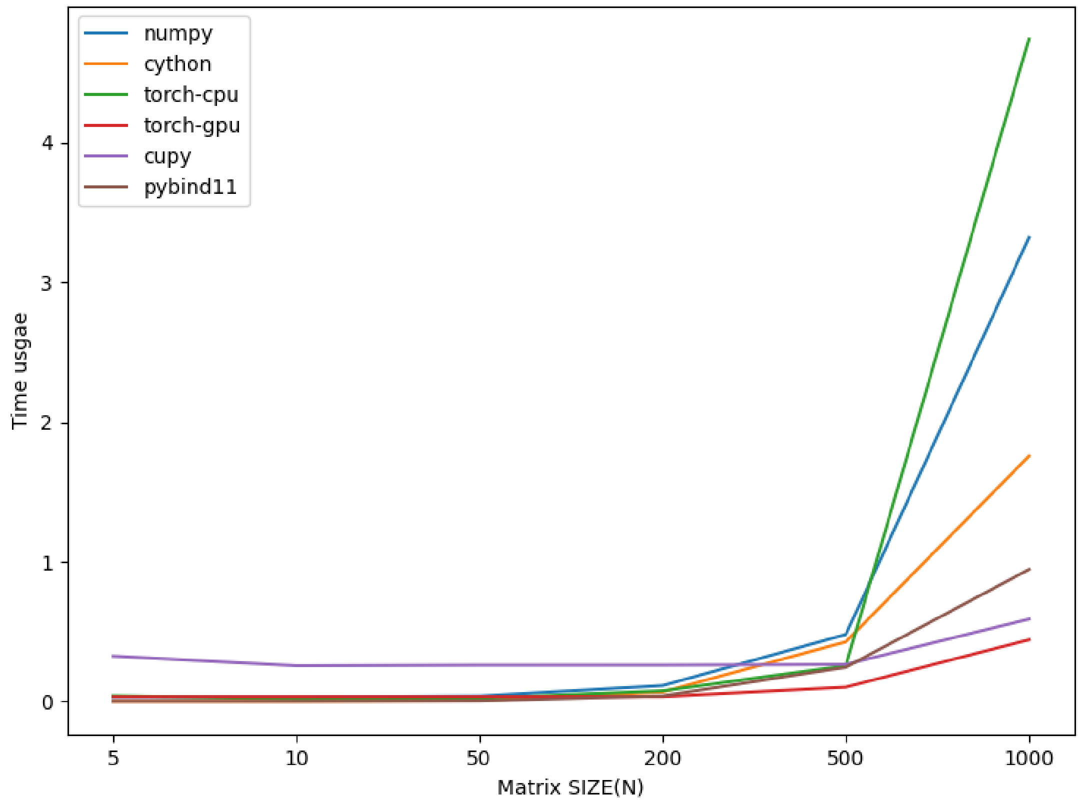
Task 2.7: Measure the performance (execution time) with GPU (PyTorch and cupy) and make a plot of the execution time varying the size the of the grid. Compare and comment of the performance difference with and without GPU.

Task 2.8: Save the newgrid matrix as an hdf5 file using h5py (see the tutorial [C.4 Tutorial - The HDF5 Library/ModuleC.4 Tutorial - The HDF5 Library/Module])

Its saved as file Assignment 3/newfs.hdf5 in github.

Bonus Task B.1: Develop the function you want to optimize in C/C++ or Fortran, make it a library, and couple C and Python. You can use the approach of your choice (see the lecture on foreign function interfaces). This is part of the bonus exercise.

2.5-2.7 & B.1: (codes in github Assignment 3/e2GaussSeidel_gpu.py)
Set grid size as [5,10,50,200,500,1000]



```
Numpy time   average:6691.56 ms  min:311.78 ms   max:33210.36 ms
Cython time  average:3762.01 ms  min:6.23 ms     max:17565.00 ms
torch time   average:8590.33 ms  min:175.52 ms   max:47430.96 ms
 torch time  average:1131.07 ms  min:331.00 ms   max:4414.00 ms
 cupy time   average:3257.80 ms  min:2558.38 ms  max:5894.48 ms
 pybind11 time   average:2056.57 ms  min:8.66 ms     max:9448.77 ms

-------------FLOPS/s--------------
```

	numpy	Cython	torch-cpu	torch-gpu	cupy	pybind11
5	7834.7597767167745	400128.0409931737	6227.006777468889	7552.053738720625	772.128965995529	287999.5207529108
10	64142.79931820587	2608759.1699005608	113928.85655562289	59219.74959700754	7817.354454971491	2086284.5566947178
50	6539761.633020976	51854923.20819614	11389705.328249888	7393722.558022292	960794.3669502253	79884506.1783074
200	140934861.80952805	235799866.7495115	210882646.41696766	467845586.8624004	61465712.84838072	403258022.37767655
500	521844765.2471712	585960167.9606569	978101167.7579098	2442111934.4951224	942993319.1750535	1022711396.9922781
1000	602220955.4731907	1138625289.0691829	421665034.350375	4530994555.978028	3392991701.334417	2116664103.4217467

As the plots shows:

1、Performance with GPU is highly better when Matrix Size is big enough and time usage is stable as Matrix Size increasing. Because of th advantage of Synchronous Computing of GPU.

2、With GPU performance of torch is better than cupy.

3、Without GPU performance is: pybind11>cython>numpy. In particular, torch in cpu is better than cython when Matrix Size is small, and no better than numpy as Matrix Size big enough.

B1: Created two libs by Cython and Pybind11, Assignment3/gauss_seidel_cython.pyx, Assignment3/gauss_seidel_c.cpp, respectively.

Bonus Exercise - Get familiar with VTK and Paraview

In this exercise, we ask you to experiment with one of the most used tools to visualize datasets coming from HPC simulations, Paraview. One of the most common and most used formats for visualization is called VTK.

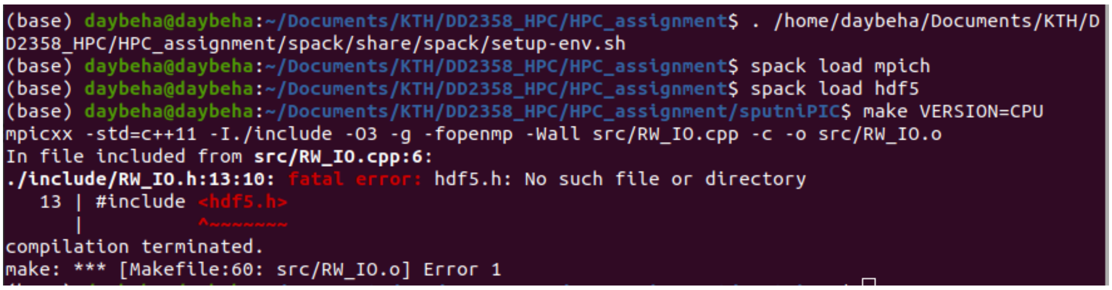
As part of the exercise, you will need to repeat all the different steps presented in the [C.6 Tutorial - VTK & Visualization with ParaviewC.6 Tutorial - VTK & Visualization with Paraview]

To install sputniPIC follow the instructions in the tutorial for paraview:

1. Install the **mpich and hdf5 libraries using spack**
2. Install the SputniPIC code
3. When you run sputniPIC, the code will create a number of VTK files in the data folder (be sure to create a directory called data otherwise, you will get an error and you can't run sputniPIC)
4. Download and install paraview on your system
5. Use paraview for the visualization

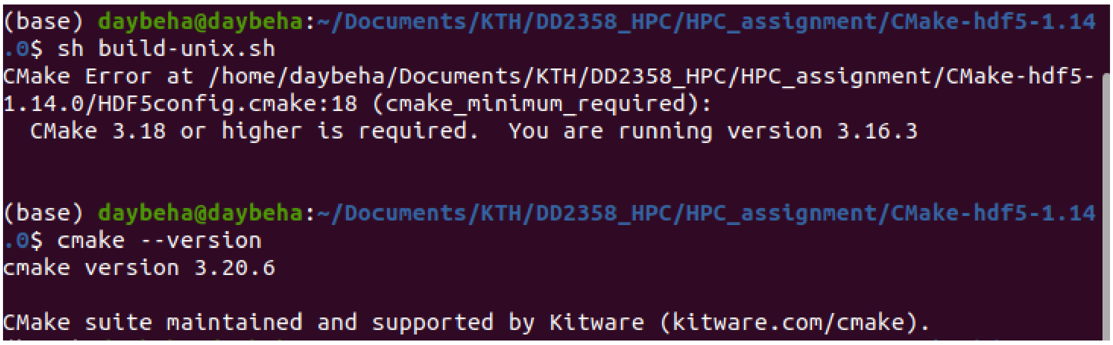
Task 3.1: Make a volume plot and a slice of the rhoe at the last step recorded in the vtk file as shown in the tutorial. Add the snapshots you took to your report.

Get error when make **sputniPIC**:



```
(base) daybeha@daybeha:~/Documents/KTH/DD2358_HPC/HPC_assignment$ . /home/daybeha/Documents/KTH/DD2358_HPC/HPC_assignment/spack/share/spack/setup-env.sh
(base) daybeha@daybeha:~/Documents/KTH/DD2358_HPC/HPC_assignment$ spack load mpich
(base) daybeha@daybeha:~/Documents/KTH/DD2358_HPC/HPC_assignment$ spack load hdf5
(base) daybeha@daybeha:~/Documents/KTH/DD2358_HPC/HPC_assignment/sputniPIC$ make VERSION=CPU
mpicxx -std=c++11 -I./include -O3 -g -fopenmp -Wall src/RW_IO.cpp -c -o src/RW_IO.o
In file included from src/RW_IO.cpp:6:
./include/RW_IO.h:13:10: fatal error: hdf5.h: No such file or directory
   13 | #include <hdf5.h>
      |          ^~~~~~~~
compilation terminated.
make: *** [Makefile:60: src/RW_IO.o] Error 1
```

And when I'm trying to configure hdf5 from the source, I got weird ouput:

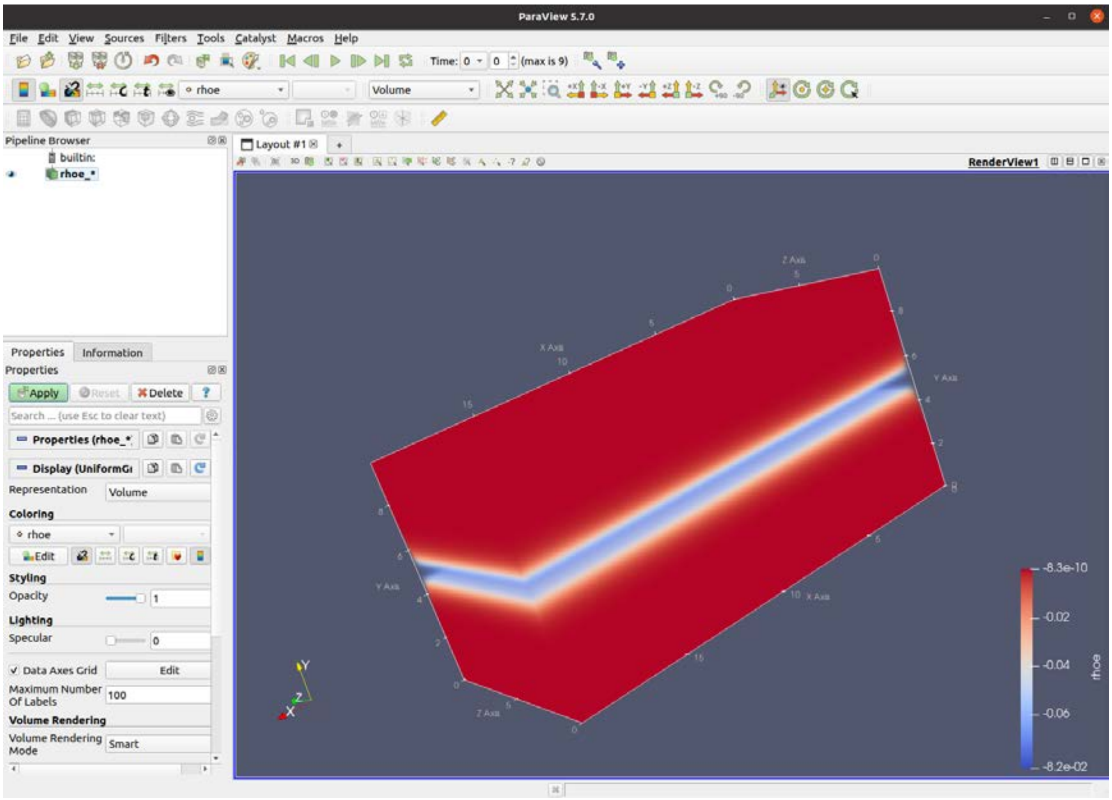


```
(base) daybeha@daybeha:~/Documents/KTH/DD2358_HPC/HPC_assignment/CMake-hdf5-1.14.0$ sh build-unix.sh
CMake Error at /home/daybeha/Documents/KTH/DD2358_HPC/HPC_assignment/CMake-hdf5-1.14.0/HDF5config.cmake:18 (cmake_minimum_required):
  CMake 3.18 or higher is required.  You are running version 3.16.3


(base) daybeha@daybeha:~/Documents/KTH/DD2358_HPC/HPC_assignment/CMake-hdf5-1.14.0$ cmake --version
cmake version 3.20.6

CMake suite maintained and supported by Kitware (kitware.com/cmake).
```

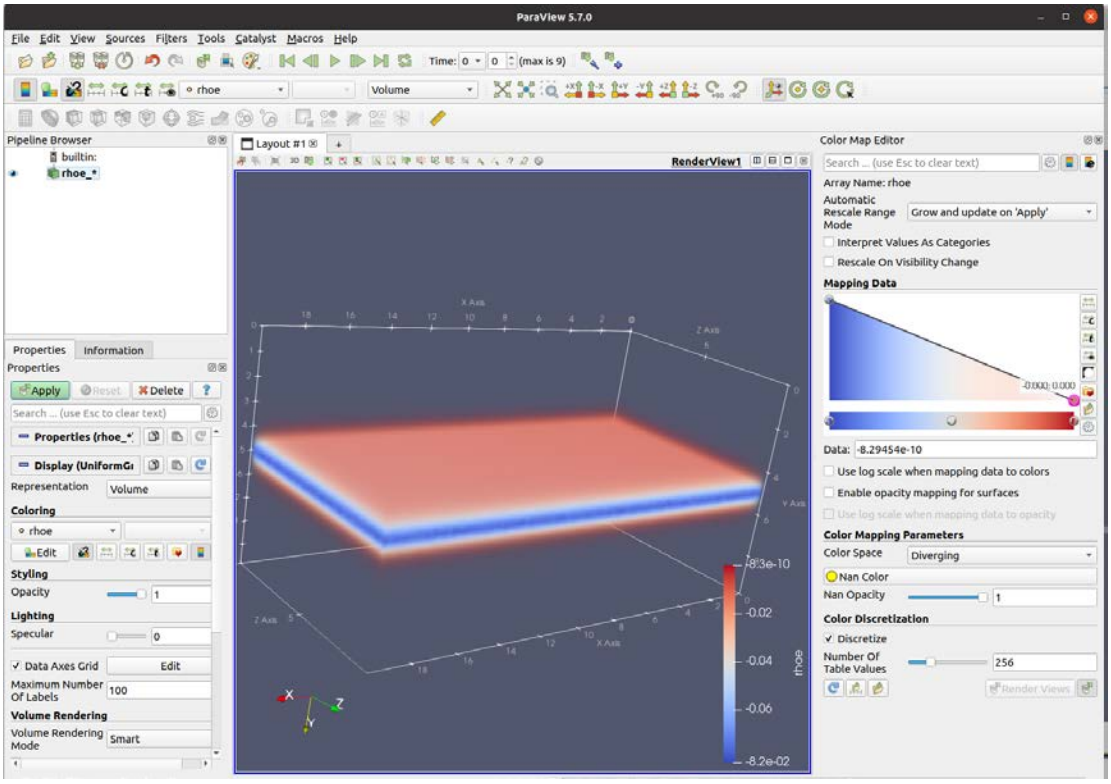
……

Lost of problems to load [hdf5]() and [mpich]() with spack using Ubuntu20.04.
Solved them by install hdf5 and mpich frome source and build with cmake.

Successfully open rhoe_*.vtk with **ParaView** using **Volume rendering:**

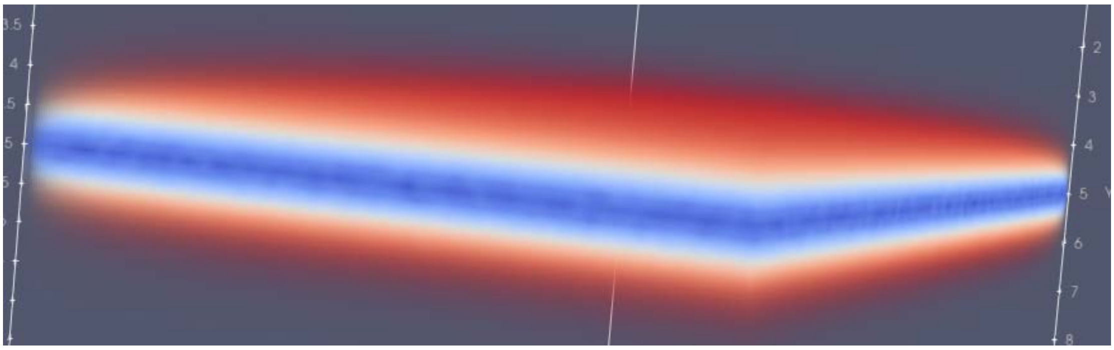


reverse the coloring of Mapping Data:

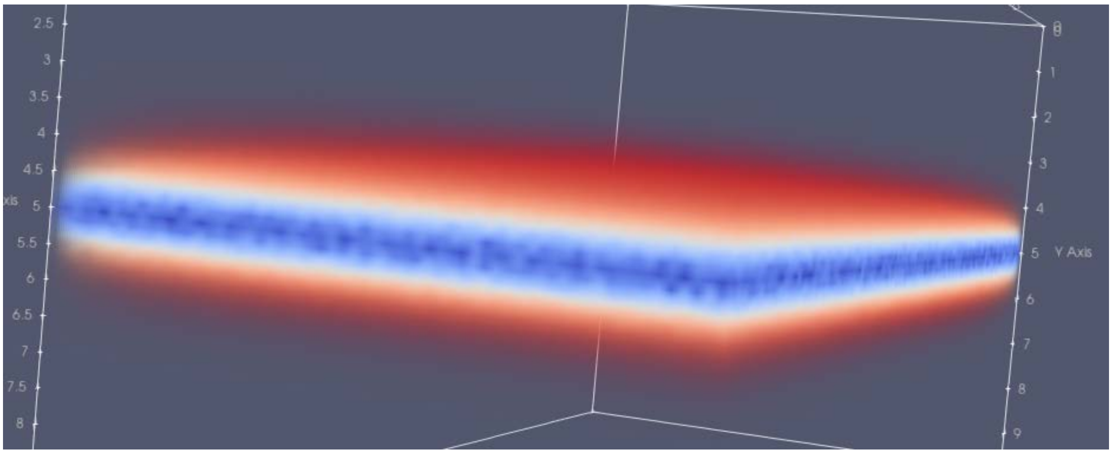


play all the simulation output time step:

time=0

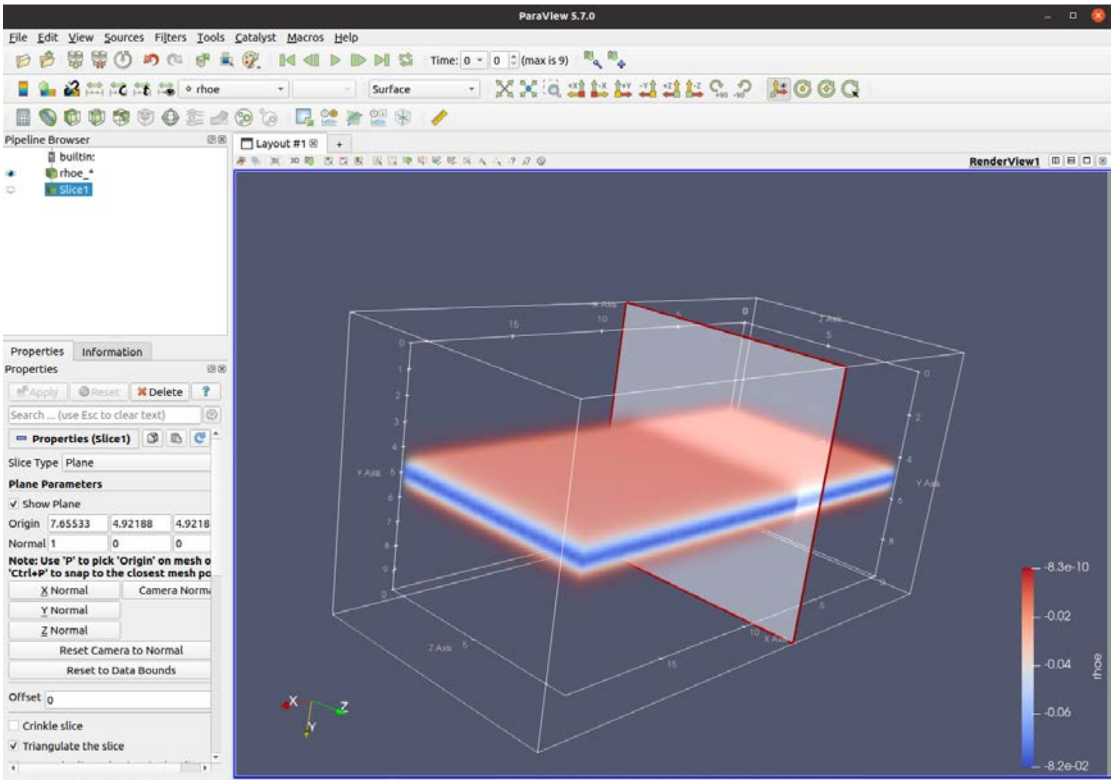


time=9 (max is 9)

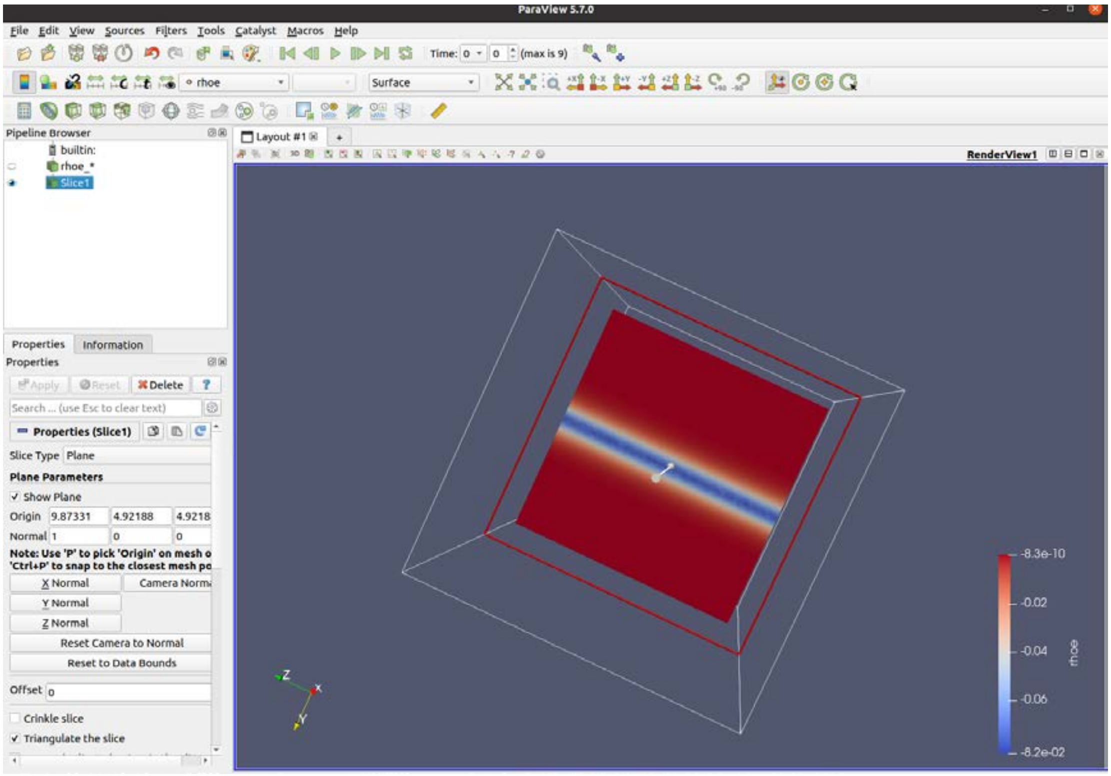


It’s different in the middle of volume. Seems clearer.

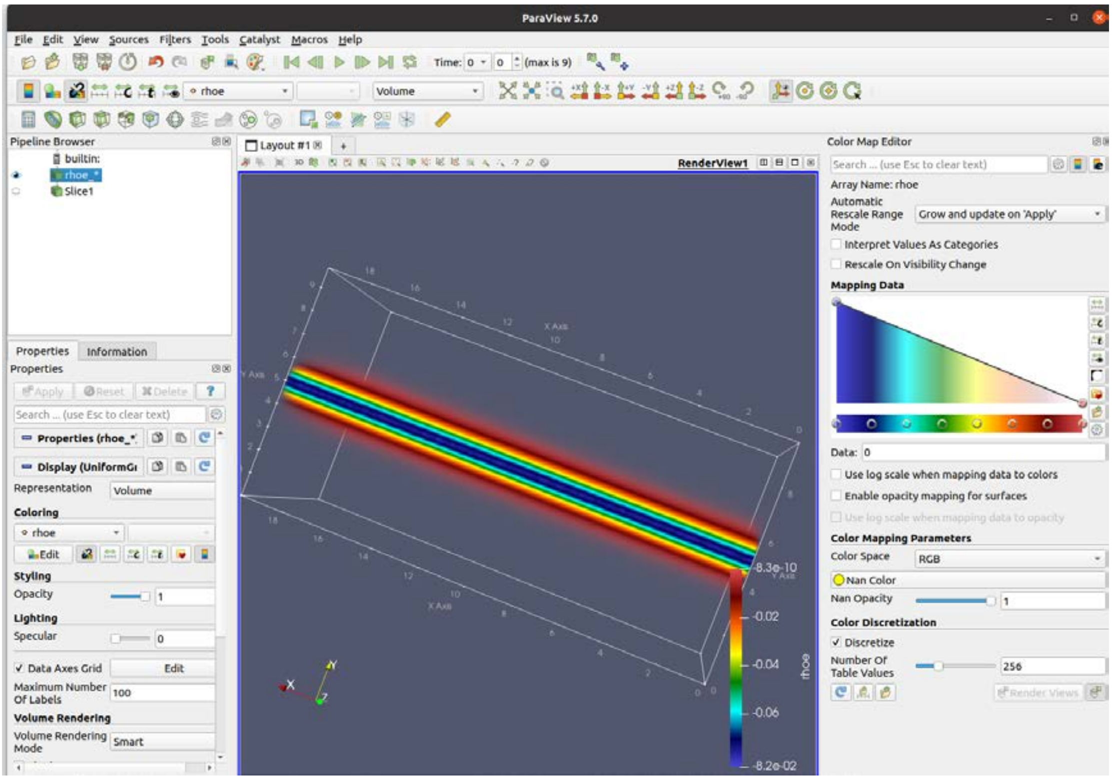
To slice data:



data sliced:



show with rainbow desaturated:


ParaView 5.7.0
File Edit View Sources Filters Tools Catalyst Macros Help
Pipeline Browser
builtin:
rhoe_*
Slice1
Layout #1
RenderView1
Color Map Editor
Array Name: rhoe
Automatic Rescale Range Mode
Grow and update on 'Apply'
Interpret Values As Categories
Rescale On Visibility Change
Mapping Data
Use log scale when mapping data to colors
Enable opacity mapping for surfaces
Color Mapping Parameters
Color Space
RGB
Nan Color
Nan Opacity
Color Discretization
Discretize
Number Of Table Values
256
Properties
Information
Representation
Volume
Coloring
rhoe
Styling
Opacity
Lighting
Specular
Data Axes Grid
Maximum Number Of Labels
100
Volume Rendering
Volume Rendering Mode
Smart
-8.3e-10
-0.02
-0.04
-0.06
-8.2e-02
rhoe
X Axis
Y Axis