DD2358 VT25 Introduction to
High Performance Computing

Project Report: Optimizing Finite-Volume Fluid Simulation

Rishi Vijayvargiya[†] rishiv@kth.se
Paul Mayer[†] pmayer@kth.se

Prefix

The code for our project can be found at this location: `https://github.com/paulmyr/DD2358-HPC25/tree/master/10_project_rishi_paul`. The original code for this project is taken from the following GitHub repository. The code is written by Philip Mocz.

We expect to get a grade of **A** (for the project and the course). Finally, we made following changes before starting with the project:

- Fixed a bug that resulted in an infinite loop when setting the `plotRealTime` flag to `False`.

Contents

[†]Authors made equal contribution to the project

1 Introduction

The Kelvin-Helmholz-Inequality is a phenomenon that arises when two fluid layers of different velocities interface with each other. It results in characteristic patterns that can be observed, e.g. in clouds, the surface of the sun or in jupyters colourful atmosphere. Philip Mocz created an which produces a visualization of the characteristic swirls of the K-H-inequality.

The Finite Volume method is a popular simulation technique to simulate fluids or partial differential equations that can be represented in a conservative form. This is often the case for equations that describe physical conservation laws. The Euler-Fluid-Equations (a simplifications of the Navier-Stokes-Equations) can be represented — besides it's primitive description — in such conservative form. Within the algorithm, Philip Mosz makes use of both representations of the formula. He uses the primitive form to extrapolate values in time and space, but uses the conservative representation to derive the update formula. Numerically, this achieves the best of both worlds. Extrapolating within the primitive form is more stable whereas the conservative representation is used to compute the update efficiently.

The algorithm in principle works as follows: For each time step and each cell do

1. Get cell central primitive variables (convert from conservative ones)
2. Calculate time step size Δt
3. Calculate gradient using primitive variables (using central differences)
4. Extrapolate primitive variables in time
5. Extrapolate primitive variables to faces
6. Compute fluxes along each face
7. Update solution by adding fluxes to conservative variables

First it is important to note, that all of these steps are computed for the density ρ, pressure p as well as the velocity in both dimensions v_x and v_y. These computations don't rely on each other and can therefore be performed in parallel.

Second, each function call in itself usually performs a computation on the whole grid. The original implementation makes use of this by vectorizing the computation (e.g. using `np.roll`). These calculations can be parallelized — they are in fact embarrassingly parallel.

We aim to introduce optimizations that decrease the overall runtime of the computation. To achieve this, we use techniques obtained from the lectures, specifically using cython to use precompiled c code to reduce python overhead and investigate a distributed computation approach utilizing dask. As a little bonus treat, we combined the dask and cython approach. Lastly, we investigated using GPU accelerators using pytorch.

2 Baseline tests

2.1 Testing and profiling setup

All runtime tests that show results for a specific grid size were computed on a 2021 M1 MacBook Pro (16') if not explicitly stated otherwise. For every grid we simulated 256 time steps without computing the visualization. Every test was run three times, runtimes were then averaged.

Some of the profiling graphs are computed on a 2020 M1 MacBook Air, we specify it explicitly when this was done so. All implementations were tested to ensure correctness by comparing the final `rho` values on the grid between baseline and optimized implementations. Every approach presented in this report passed this test.

2.2 Runtime

We approached this problem by trying to get a basic understanding of what might be the runtime bottleneck. For this we used the line profiler. We simulated[1] on a 256x256 grid for 256 iterations. We

[1] using the MacBook Air (2020)

came to the conclusion that a large portion (nearly 50%) of computation time is spent to compute the fluxes (`getFlux` method).

```
Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
   195                                           @profile
   196                                           def main():
[...]
   272                                               # compute fluxes (local Lax-Friedrichs/Rusanov)
   273       256     357351.0   1395.9     25.0      flux_Mass_X, flux_Momx_X, flux_Momy_X,
                                                        flux_Energy_X = getFlux(...)
   274       256     320673.0   1252.6     22.4      flux_Mass_Y, flux_Momy_Y, flux_Momx_Y,
                                                        flux_Energy_Y = getFlux(...)
[...]
```

All other function calls accumulated to the other 50%, however each in itself were more or less neglegtable (at most below 4%). This strongly incentivised to focus our efforts onto this part of the computation.

The runtime of the simulation mostly depends on two parameters. First and foremost the number of time steps: Each time step is computed in one iteration of the main loop. Figure 1 displays the runtime the computation of each time step needs. We can observe that the runtime per iteration is mostly consistent over the course of the simulation.

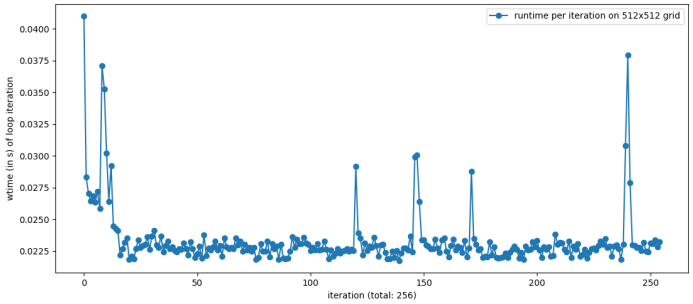


Figure 1: Baseline Runtime per iteration on 2021 MacBook pro

Important note: To simulate a fixed time, the total number of time steps increases by increasing grid resolution. With other words, simulating two seconds using a fine grid uses more iterations than simulating two seconds using a coarse grid. The reason is that dt, which is computed each loop, is dependent on dx — the granularity of the mesh.

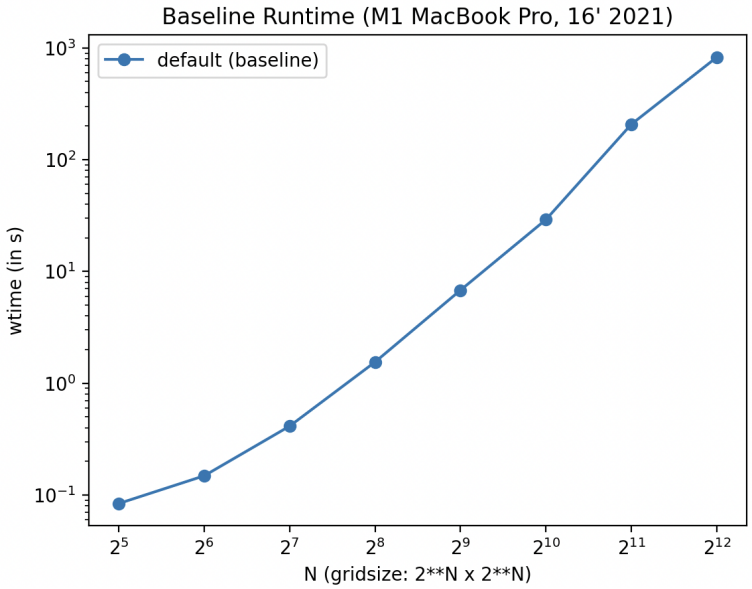


Figure 2: Baseline Runtime per iteration on 2021 MacBook pro

The second important parameter is the grid size. Finer grids allow for higher precision but increase the computational complexity quadratically. To measure runtime over a variety of grid sizes, we decided to keep the number of time steps constant. We simulated 256 time steps over all grid sizes and performed each computation three times, which we then averaged. The *baseline/default*[2] is the original implementation of the simulation. Figure 2 visualizes the wall times for different grid sizes of the baseline implementation on a log-log scale.

2.3 Memory

We measured the memory consumption of the simulation over the runtime on a 512x512 grid. Figure 3 depicts the measurement we obtained using the mprof module in python. We see that once all memory has been allocated, all computations are performed in-place. This is good, since the reallocating memory and waiting for the garbage collector to free memory could lead to performance issues, especially on larger grids.

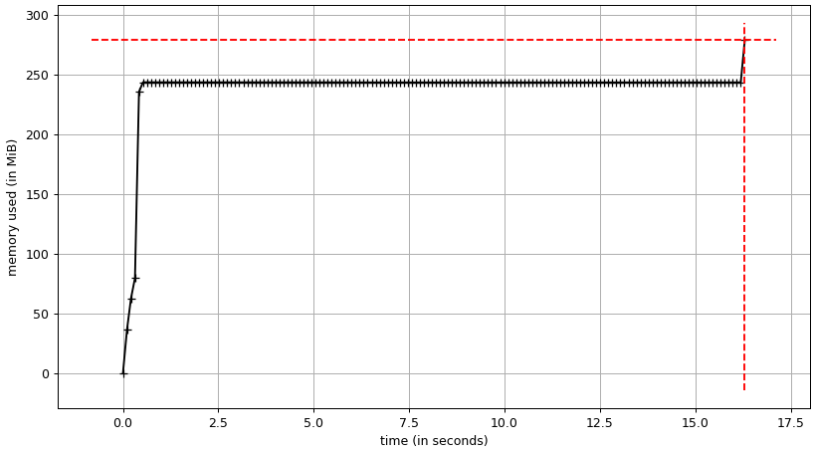


Figure 3: Memory consumption 512x512 grid measured on M1 MacBook Air

3 Optimizations using Cython

Since `getFlux` was the bottleneck for the simulation based on our profiling, we focused on optimizing this function using Cython. We investigated four approaches in total, only one performed better than the baseline[3]. All runtimes presented in this section timed the simulation using different versions of the `getFlux` method.

3.1 Attempt 1: Using *np.asarray* With Minimal Changes

For the first attempt, we extensively use Cython's *typed memoryviews*; both for arguments and intermediate variables defined in the function. The memoryviews are converted to numpy arrays using `np.asarray` to allow for vectorized operations. This way there is minimal deviation from the structure of the original code[4]. Figure 4 depicts the performance of this attempt compared to the baseline, which is slightly worse. We believe this can be explained by the frequent calls to `np.asarray`, which likely requires interactions with the Python runtime. The original code was vectorized incredibly well, so there were no easy performance gains (e.g. by optimizing nested `for`-loops) that Cython could have automatically achieved without our intervention.

[2]Code for baseline found here.
[3]The code for this section is present in the *cython/* directory of the repository here
[4]Code found here (*getFluxAsArray*).

3.2 Attempt 2: Re-Implement Vectorized Arithmetic

We re-implemented[5] numpy's vectorized operations using simple nested `for`-loops. For example, the addition of 2 numpy arrays `a + b` was re-implemented with nested `for`-loops using element-wise additions. We continued using typed memoryviews in all places. We hoped that Cython would be able to optimize the simpler nested loops, leading to a better runtime; however, this was not the case. The comparison of this attempt with the baseline and Attempt-1 can be seen in Figure 4.

One explanation for the poor performance of this implementation could be attributed to the fact that each operation requires the program to iterate over the arrays multiple times. The original code contains various combination of such simpler operations to compute the intermediate variables. Another issue could be the fact that numpy is very good at vectorizing these optimizations, whereas the used c-compiler of cython might not.

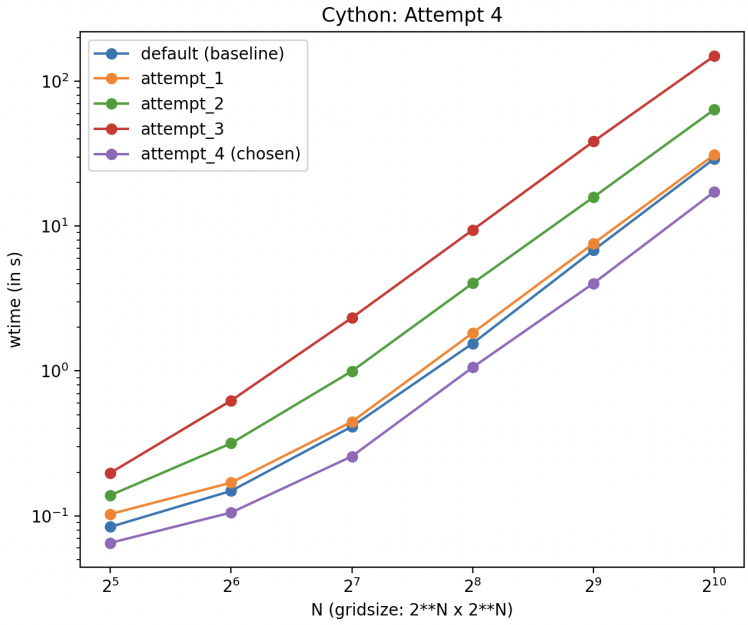


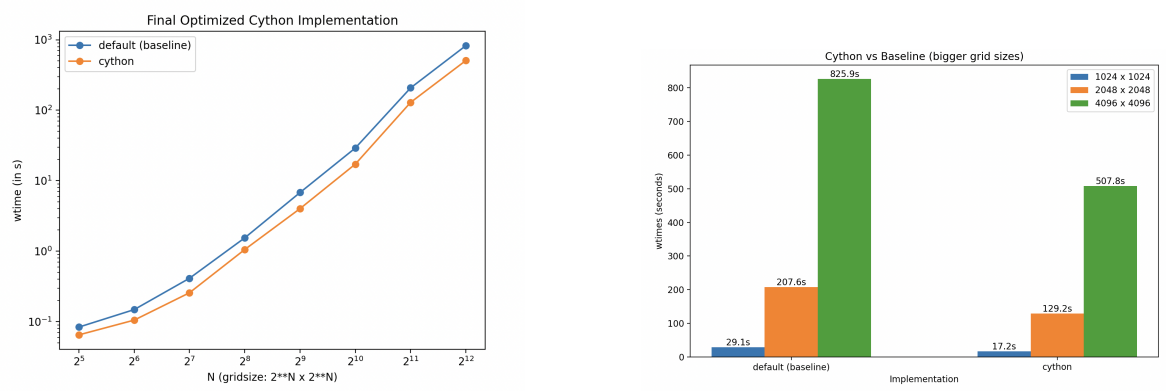
Figure 4: Attempts 1-4 vs Baseline

3.3 Attempt 3: Single Python Loop

We observed that all the computations for the returned fluxes can be performed in an *embarrassingly parallel* manner: no cell of the returned grids dependent on any other cell. Thus, iterating over the input grids one cell at a time and computing the resulting fluxes would give us the same result as the vectorized version. We implemented [6] a *single* nested loop in Python, where we compute each flux component one cell at a time using typed memoryviews. We disabled some cython flags (such as `boundscheck`, `wraparound`) to help with speed.

As depicted in figure 4, the performance was the worst out of all attempts. This was surprising to us: we believed that Cython would be able to minimize the number of calls to the Python runtime because the nested loops involved only arithmetic operations on elements of typed memoryviews. However, on examining the HTML file generated using the `cython -a ...` command, we still observed heavy Python runtime calls for computations, particularly those that involved accessing elements of the typed memoryviews (see Figure 13). The frequent callbacks to the Python runtime must have reduced the number of optimizations that Cython was able to make — leading to worse performance.

[5] Code found here (*getFluxAsLoops*).
[6] Code found here *(getFlux)*.



(a) Line Plot (b) Bar Graph

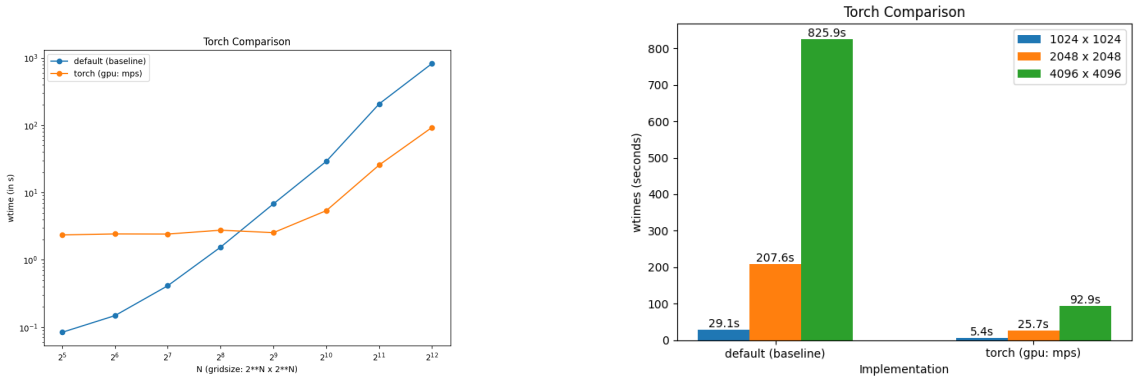
Figure 5: Cython Attempt 4 vs Baseline

3.4 Attempt 4 (Chosen Optimization): Single C Loop

Since Cython was not able to prevent the Python runtime interactions, we decided to reimplement[7] the core logic of the original `getFlux` method in C directly. The syntax and structure of the implementation was inspired by the example presented in the Cython documentation (link). The numpy arrays passed to the `getFluxRawC` are in contiguous row-major (C) order, which makes the port of the python code to C easier. Correctness is tested in the `flux_test.py` file to verify similarity between the default `getFlux` and `getFluxRawC`. We still use typed memoryviews for the numpy arrays, along with disablement of some Cython flags.

The runtime-comparison of this approach with the other attempts and baseline is present in Figure 4. Of all attempts, this approach performed the best and most importantly: better than the baseline for all grid sizes. This may indicate that performing everything in one-loop is the right approach and that Cython's extensive calls to the Python runtime limited runtime optimizations in Attempt 3. In Figure 5a, we see that the performance of this approach remains better than the baseline for grid sizes 2048 and 4096. The bar graphs in Figure 5b highlights the performance difference of this Cython optimization over the baseline.

4 Optimizations using Pytorch



(a) Line Plot (b) Bar Graph

Figure 6: Runtime pytorch implementation vs baseline

The original implementation makes heavy use of numpy functionalities. All data is stored as numpy arrays that are incrementally updated using the computed fluxes. This provides an easy

[7] Code found here *(getFluxRawC)*. The C code is found here.

approach to move the computations on a GPU using pytorch[8]. Pytorch's computations are run using tensors — an immutable data-type that shares resemblance with numpy arrays. A big difference however is that tensor operations are supported to be computed using accelerators. Due to the fact that pytorch mirrors numpys interface, we simply need to convert said numpy arrays to pytorch tensors and tell pytorch to perform the computations using GPU compatible hardware.

To make runtime results more comparable to the other approaches, we used the M1's metal, the integrated GPU on the M1 chip. This is achieved by specifying the `mps` device in torch. The only difference to CUDA is that metal does not support 64bit floats. Otherwise, the handling is identical.

As one can see in figure 6, making use of accelerators *greatly* improves the overall runtime for larger grid sizes. However, on smaller grid sizes, the overhead significantly slows the computation.

5 Optimizations using Dask

We approached optimization with Dask from two angles[9]. For all dask runs, the default scheduler settings were used, which on the Mac book Pro gave us *5 workers, 2 threads per worker*.

Each iteration of the simulation is dependent on the outputs of the previous iteration. Coming up with Dask optimizations that delayed the call to `compute()` till the last possible moment (preferably outside the loop), was difficult. Particularly, the termination condition of the original code dependent on the time parameter `t`, which was updated in every iteration based on the value of `dt` that was computed inside the loop. Thus, the 2 optimizations we tried respect this starting structure of the code, and consequently `compute()` is called *in each iteration*. While fixing the number of iterations could generate better results, we were still able to obtain promising improvements.

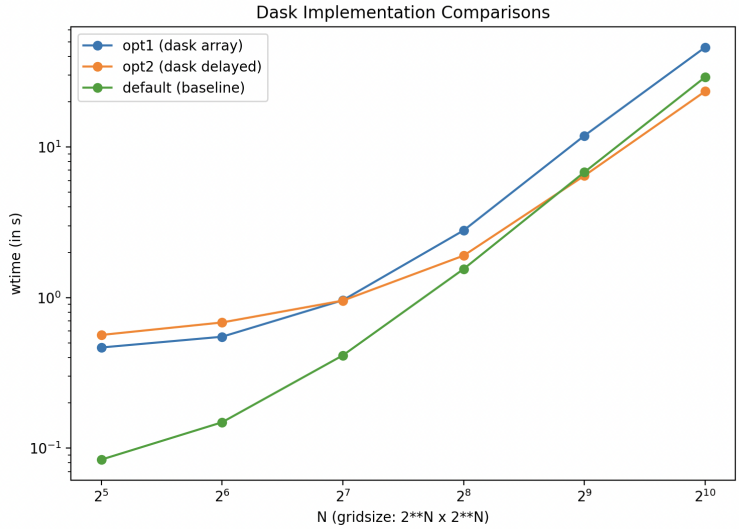


Figure 7: Dask Attempts 1-2 vs Baseline

5.1 Attempt 1: Dask Arrays

As noted above, `getFlux` is the bottleneck for the baseline simulation. This method is *embarrassingly parallelisable* (no cell computation dependent on any other cell). We used `map_blocks` from `dask.arrays` on the `getFlux` for the `getFlux` method[10]. We only used this for the computation of the fluxes in `x-` direction first, hoping to extend it to `y-` direction if the performance was good. We created dask arrays for the input to `getFlux` in each iteration. We then called `compute()` on the value returned by `map_blocks` to obtain the fluxes in `x-` direction in each loop iteration. The task-stream visualizations were done while running the experiment on a grid size of 128 till 2 seconds of *simulated time*.

[8]The code for this section is present in the *torch/* directory of the repository here.
[9]The code for this section is present in the *dask/* directory of the repository here.
[10]Code found here.

(a) Work Stream (64 chunks)

(b) Work Stream (4 chunks)

Figure 8: Work Stream for Dask Attempt 1

We varied the chunk sizes of the dask arrays to see what impact they have on the runtime. A plot of this can be seen in Figure 14. In Plot 7 we present only the runtime for the best performing chunk-size, which was 1. In summary, the performance decreased with an increase in chunk-sizes.

The performance of this implementation always performed worse than the baseline. Several factors could have contributed to this. Firstly, there is overhead involved in Dask having to create the dask arrays with the correct chunk sizes, allocating them to the workers, setting up the scheduler, etc. This overhead can be present in the form of communication between workers. We observed this on the task-stream for the computation on the Dask Dashboard. The red/dark-red observed in the task stream in Figure 8 indicates a lot of transfer between workers[11]. This was also present when we had fewer chunks than cores/workers (Figure 8b). Additionally, the call to compute in each loop iteration would also lead to a slowdown.

We thought of some ideas to address some of our concerns here (such as trying to delay `compute` for everything except `dt`). However, our experience with Assignment 4's Bonus lead us to believe that even with fewer `compute()` calls, the performance of dask-arrays in a single-machine environment would likely still be worse than the baseline (because of the set-up overhead discussed above). Thus, before doing this, we tried a different approach: using `dask delayed`.

5.2 Attempt 2 (Chosen Optimization): Dask Delayed

Instead of focusing only on the parallelisability within `getFlux`, we observed that there were several calls of helper functions that were independent of each other. For instance, computing `getFlux` in x- and y-direction can be done independently. By looking at the bigger picture, we decided to use dask-delayed to store the delayed tasks for each helper function in an array and then called `dask.compute()` for each element of the task array[12].

Each such helper function was tagged the `@delayed` decorator. The performance of this imple-

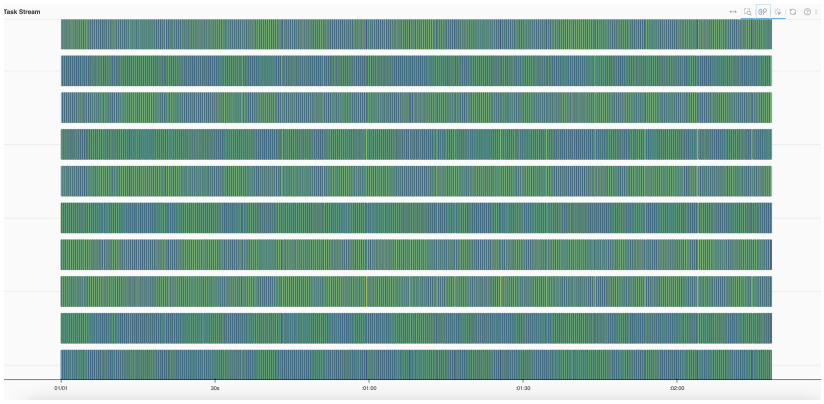


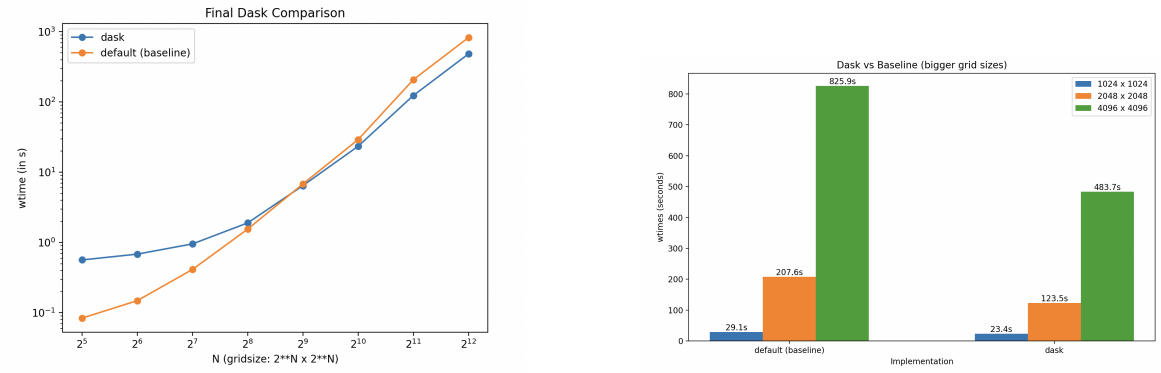
Figure 9: Task Stream for Dask-Delayed Optimization

mentation with Attempt-1 (dask-arrays) and baseline can be seen in Figure 7. We see that initially for very small grids, this attempt performs worse. This is attributed to the increased number of

[11]The original html file is found here.

[12]Code found here.

`compute()` calls for each independent helper-function, which requires all parallel calls to complete before progressing. However, as the grid sizes increase, this approach comes out on top. This indicates to us that as the grid-size increases, the sheer number of computations to be performed *outweighs* the cost of the multiple `compute()` calls (and other overhead required by dask-delayed): leading to a better performance when doing the computations in parallel for grid sizes bigger than 512. The improved performance compared to Attempt-1 could be explained by almost no interaction required between workers, as can be seen by the absence of red in the task-stream graph in Figure 9. This was our chosen Dask optimization. In Figure 10a, we see that the performance of this optimization



(a) Line Plot (b) Bar Graph

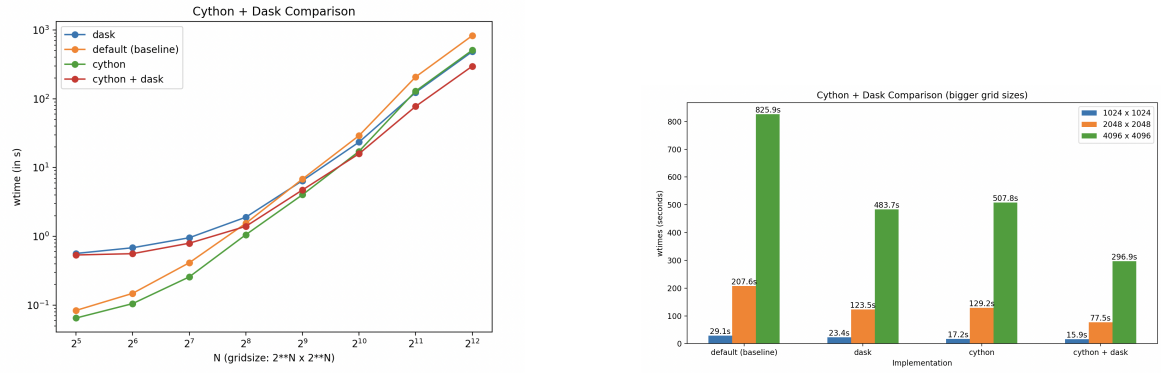
Figure 10: Dask Attempt 2 vs Baseline

remains better than the baseline for larger grids, as we hypothesized. Figure 10b highlights the time-difference that might be lost in the log-scale based line plot.

6 Bonus Optimization: Cython + Dask

We observed that for the 2 largest grid sizes (2048 and 4096), our Cython optimization performed *worse* than our Dask optimization. To us, this indicated that any performance advantage obtained from performing the time-consuming `getFlux` operations in C was lost when it came to the sheer number of operations to be performed, thus giving our parallel Dask optimization an advantage.

As a bonus, we decided to combine the two: we performed the `getFluxRawC` computation (our Cython optimization) in parallel, using our Dask-Delayed optimization[13].



(a) Line Plot (b) Bar Graph

Figure 11: Cython + Dask Awesomeness

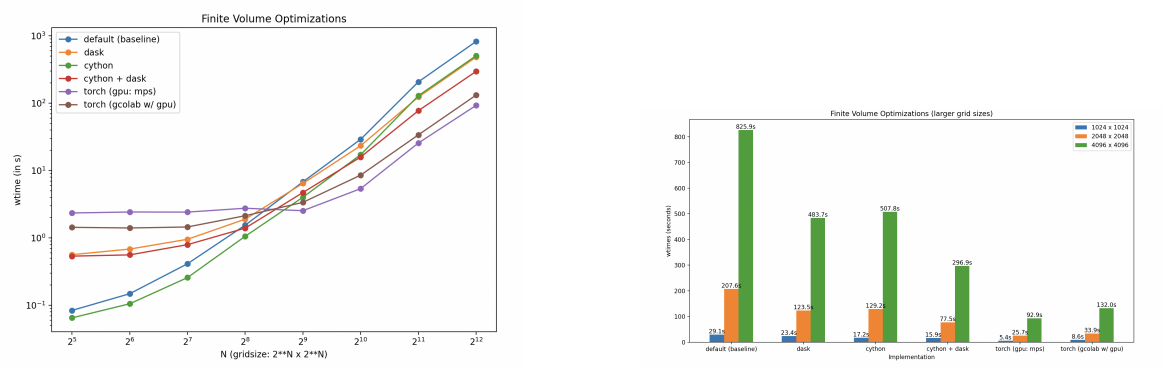
This lead to incredible results for larger grids, as can be seen in 11a. The combined approach outperforms the approaches of the individual optimizations, and of the baseline. The bar graph in 11b shows the runtimes in a non-logarithmic scale, highlighting the performance boost. For smaller

[13]Code found here.

grids, this appears to be between the Cython and Dask optimizations. This could be explained by the fact that the dask-overhead *pulls up* the runtime compared to the sole Cython optimization, and the use of the optimized `getFluxRawC` *pulls down* the runtime compared compared to the sole Dask optimization.

7 Conclusion

The final plot of baseline runtime, and of all the *chosen* optimizations, can be seen in Figure 12a. Figure 12b shows the bar graphs for the largest 3 grid-sizes to highlight the performance improvements over the baseline. The trends and likely explanations for the trends observed in the runtime compared



(a) Line Plot

(b) Bar Graph

Figure 12: Concluding results

to the baseline have been discussed extensively in their respective sections. Here, we would like to make one final important observation about the general trends. For optimization approaches that require overhead, whether it be initial or otherwise (e.g., moving data to the GPU, Dask-Delayed setup, `compute()` calls, etc), struggle to perform on coarse grid resolutions. There, techniques without significant overhead perform better. However, the advantages of overhead starts becoming apparent with increased grid-sizes. This is when the sheer number of computations becomes overwhelming It helps the runtime to prepare for such computations to perform them in parallel — as can be seen by the better performance of the Dask, Dask with Cython, and the Torch implementations on larger grids.

8 Appendix

8.1 Cython Attempt 3: Python Runtime Interactions

```
cdef unsigned int N = len(rho_L)

cdef double[:,:] flux_Mass = np.zeros((N, N), dtype=np.double)
cdef double[:,:] flux_Momx = np.zeros((N, N), dtype=np.double)
cdef double[:,:] flux_Momy = np.zeros((N, N), dtype=np.double)
cdef double[:,:] flux_Energy = np.zeros((N, N), dtype=np.double)

cdef unsigned int i, j

for i in range(N):
    for j in range(N):
        en_L = P_L[i, j]/(gamma-1)+0.5*rho_L[i, j] * (vx_L[i, j]*vx_L[i, j] + vy_L[i, j]*vy_L[i, j])
        en_R = P_R[i, j]/(gamma-1)+0.5*rho_R[i, j] * (vx_R[i, j]*vx_R[i, j] + vy_R[i, j]*vy_R[i, j])

        rho_star = 0.5*(rho_L[i, j] + rho_R[i, j])
        momx_star = 0.5*(rho_L[i, j] * vx_L[i, j] + rho_R[i, j] * vx_R[i, j])
        momy_star = 0.5*(rho_L[i, j] * vy_L[i, j] + rho_R[i, j] * vy_R[i, j])
        en_star = 0.5*(en_L + en_R)

        P_star = (gamma-1)*(en_star - 0.5*(momx_star**2+momy_star**2)/rho_star)

        # flux_Mass[i, j] = momx_star
        # flux_Momx[i, j] = momx_star**2/rho_star + P_star
        # flux_Momy[i, j] = momx_star * momy_star/rho_star
        # flux_Energy[i, j] = (en_star+P_star) * momx_star/rho_star

        C = max(
            math.sqrt(gamma*P_L[i, j]/rho_L[i, j]) + abs(vx_L[i, j]),
            math.sqrt(gamma*P_R[i, j]/rho_R[i, j]) + abs(vx_R[i, j])
        )

        flux_Mass[i, j] = momx_star - (C * 0.5 * (rho_L[i, j] - rho_R[i, j]))
        flux_Momx[i, j] = (momx_star**2/rho_star + P_star) - (C * 0.5 * (rho_L[i, j] * vx_L[i, j] - rho_R[i, j] * vx_R[i, j]))
        flux_Momy[i, j] = (momx_star * momy_star/rho_star) - (C * 0.5 * (rho_L[i, j] * vy_L[i, j] - rho_R[i, j] * vy_R[i, j]))
        flux_Energy[i, j] = ((en_star+P_star) * momx_star/rho_star) - (C * 0.5 * (en_L - en_R))

return flux_Mass, flux_Momx, flux_Momy, flux_Energy
```

Figure 13: Python Runtime Interactions for Cython Attempt 3

8.2 Dask Attempt 1: Dask Arrays Chunk Sizes Variation

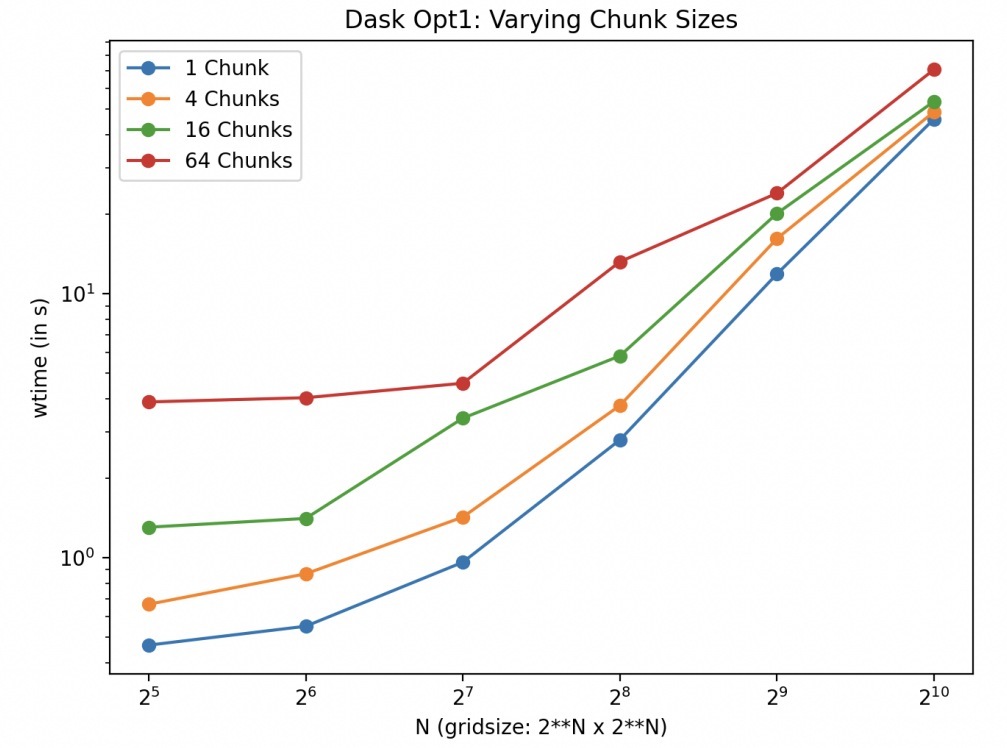


Figure 14: Dask Attempt 1: Chunk Size Variation

8.3 Runtime variation

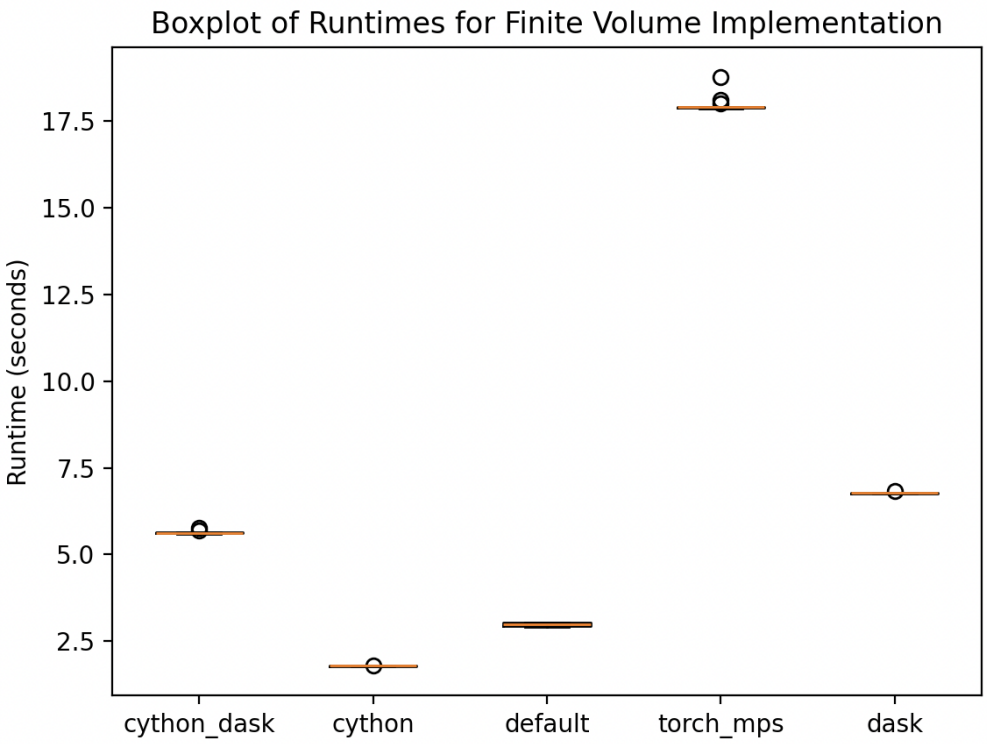


Figure 15: Runtime variation of different implementations

For completeness, we explored how volatile the runtimes were for different implementations. However, on the MacBook Pro we could not find any indication that some implementation varied significantly more than others. Figure15 shows a box-plot over runtimes simulating 2s, each performed 20 times using a grid of 128x128. As we can see, runtimes remained consistent, we did not pursue this further.