Python API:s for Running FEM

William Johansson

January 2026

Supervisor: Fredrik Larsson

1 Disclaimer

I originally had two partners, but they left late in the course and so I will describe what my work was: I implemented the simulations in Numpy, Tensorflow and Cupy, as well as helped with Numba Cuda. I standardized all the implementations before testing, and conducted all the testing and analysis myself.

2 Introduction

Optimizing simulations is very important, an increase in performance means a opportunity to increase the fidelity of the simulation. While optimization gains will come from changes one makes to the code; which changes give improvements and the size of the improvements will depend on which API one uses. So knowing which API gives the most optimization for a given application is important, and this is what I will be investigating for Finite Element Method simulations in this report.

3 Background

Our implementation of FEM contained a large amount of Lines of Code that were needed for the setup of the simulation, for exampel loading and parsing the meshes. Since the setup took a negligible amount of time, we almost exclusively focused our efforts on the simulation loop. Hence, I will focus this report on the API:s performance on the simulation loop.

The calculations that we need to compute can be described by the following pseudo code:

Listing 1: FEM Pseudo Code

```
while t < target_t
    da = dt*v - ((S@a) * (dt*dt/2))
    da = da * constrained

    v = (2/dt * da) - v
    a = da+a

    t = t + dt
```

Here **b** is a vector that contains the displacement on each Degree of Freedom (DOF). **S** is a matrix that determines the acceleration applied to each DOF based on the displacement on each DOF. **v** is a vector that contains the velocity applied to each DOF. Finally the **constrained** vector contains a 0 or a 1 for each DOF for if it should be constrained or not.

This code is embarrassingly parallelizable with one thread per DOF. Each DOF only depends on the previous states of **a** and **v**. Furthermore, each loop cycle (which I

will refer to as Tick) is very computationally cheap. I suspect that the most expensive part of each Tick would be the **S** matrix multiplication of **a** as, even though it should only need 6 multiplications due to the sparsity of **S**. The random accessing of **a** would take a comparatively large amount of time, especially with the larger tests sizes, where cache misses will be more frequent.

The API:s that i decided to test were Numpy, Tensorflow, Numba Cuda, and Cupy. Numpy, Tensorflow, and Cupy provide datastructures and functionality for matricies and so are relatively easily used to convert the mathematical notation into code. With Tensorflow and Cupy the integration with the GPU is automatically handled, with it requiring little additional cognitive load as compared to writing for the CPU. Numba Cuda on the other hand requires one to write their own kernels for the GPU and with less pre-made functionality. For example, with Numba Cuda we had to decompose our sparse matrix **S** into its coordinates and values, and then compute the matrix multiplication ourselves in the kernel.

To help with the analysis of the results I have compiled the formula below. I believe that the run time for any CPU or GPU would be roughly consistent with it:

$$RunTime = k * max(\frac{DOF}{Cores}, 1) + DriverOverhead \tag{1}$$

In the formula, k represents how long a single core would take to run a tick of the simulation for a single DOF, and would vary with performance statistics such as TFlops and memory speed. The division of DOF by Cores is limited to a minimum of 1 since if there are less degrees of freedom than cores then one core will be used per degree of freedom, and the rest will remain unused and so the runtime should stop decreasing at DOF equals Cores. DriverOverhead comes from the time spent on sending instructions between the CPU and GPU, and since every instruction should only need to be sent once per tick, it should be independent of DOF and Cores.

4 Method

I decided to compare the API:s as like for like as possible, where in I would make the programs use as similar functions and datastructures as possible between the different API:s. While I believe that a strengths to strengths comparison would be more useful in deciding which API to use, I did not believe that I had the expertise to make such a comparison useful.

Below we can see the cleaned up code for the simulation loops for the different API:s:

Listing 2: Numpy simulation loop

```
for itime in range(ntime):
    a_new = a_last + dt*v_old - S_premultiplied@a_last
```

```
    a_new = a_new * constrainment

    v_new = 2/dt * (a_new-a_last) - v_old

    a_last = a_new
    v_old = v_new
```

Listing 3: Tensorflow simulation loop

```
for itime in range(ntime):
    a_new = a_last + dt*v_old -
        sparse.sparse_dense_matmul(S_premultiplied, a_last)
    a_new = a_new * a_constraint

    v_new = 2/dt * (a_new-a_last) - v_old

    a_last = a_new
    v_old = v_new
```

Listing 4: Numba simulation loop

```
@cuda.jit
def tick(a, row_ptr, col_ind, S, v, dt, a_new,
    v_new, constrainment):
    i = cuda.grid(1)
    if i >= a.size:
        return

    if constrainment[i] == 0:
        a_new[i] = 0
        v_new[i] = 0
        return

    start_idx = row_ptr[i]
    end_idx = row_ptr[i+1]
    sum_S = 0.0
    for j in range(start_idx, end_idx):
        sum_S += S[j] * a[col_ind[j]]

    a_new[i] = a[i] + dt * v[i] - sum_S
    v_new[i] = 2 / dt * (a_new[i] - a[i]) - v[i]

threadsperblock = 64
blockspergrid = (a.size + (threadsperblock - 1))

for itime in range(ntime):
        tick[blockspergrid, threadsperblock](
```

```
            a_device, row_ptr_device,
            col_ind_device, S_premultiplied_device,
            v_device, dt, a_new_device,
            v_new_device, is_constrained_device)

        a_device, a_new_device = a_new_device, a_device
        v_device, v_new_device = v_new_device, v_device
```

Listing 5: Cupy simulation loop

```
for itime in range(ntime):
    a_new = a_last + dt*v_old - S_premultiplied@a_last
    a_new = a_new * a_constraint

    v_new = 2/dt * (a_new-a_last) - v_old

    a_last = a_new
    v_old = v_new
```

The implementation of the simulation loops are almost identical for Numpy, Tensorflow and Cupy, with the difference being in their implementations of the datastructures and operations.

The code was ran on a remote desktop computer, containing a RTX 3070 which has 5888 CUDA cores and 8 GB of memory. The time required to run the simlation was recorded using cProfiler. The API:s were tested on different mesh sizes, so that performance scaling could be observed. The tests were all conducted in a single session and were done sequentially, with the first tests being discarded as warm up.

5 Results

DOF:s	Numpy	Tensorflow	Numba Cuda	Cupy
132	4s	84s	30s	25s
462	5s	85s	30s	26s
1722	9s	84s	30s	27s
10302	36s	88s	31s	26s
40602	123s	89s	29s	26s
161202	567s	117s	45s	39s

Table 1: Table showing run time for the different API:s over the mesh sizes.

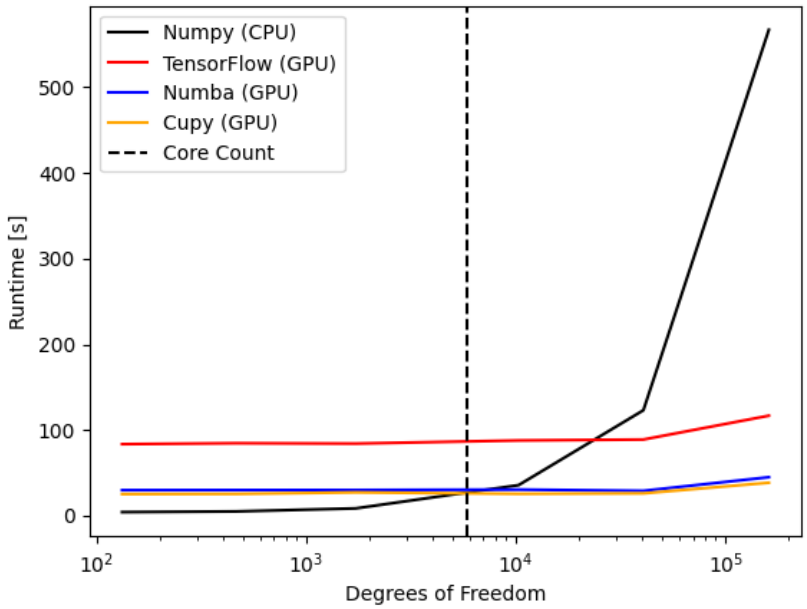

Figure 1: Graph showing how the required run time for the simulation grows with the amount of Degrees of Freedom for different API:s. GPU Core count is marked with vertical dashed line.

6 Analysis

If formula (1) is correct then it would imply that the driver overhead for the CPU with Numpy is close to zero, which is what we would expect, with the times growing linearly with DOF. The GPU times are harder to analyze due to the smaller values with high variance and the times being assumed to be constant until passing one DOF per core. However we can attempt to get a rough value on how much of the time is consumed by driver overhead versus computations. If we insert the values for Tensorflow at 1722 and 161202 DOF:s from table 1 into formula (1) then we get the following equation system:

$$84 = k * (1) + DriverOverhead$$
$$117 = k * (\frac{161202}{5888}) + DriverOverhead$$

If this system is solved then one gets that k is roughly 1.25 and DriverOverhead is roughly 82.75. This would mean that for every 5888 Degrees of Freedom, the run time increases by 1.25 seconds, on top of a 82.75 second base run time. While there

is a lot of variance the scales should still be indicative of the true values. This shows that for these mesh sizes, driver overhead can cause major performance losses, and it also implies that Numba Cuda and Cupy are significantly better at minimizing driver overhead.

7 Conclusion

For larger mesh sizes Cupy and Numba Cuda are the clear choices, with them being considerably faster. My personal recommendation is Cupy due to its similarity to Numpy and ease of use.