Overview

- We should be using graphics cards (GPUs) in classical simulations of amplifiers, couplers, etc.

- GPUs allow parallel sweeps in parameter space -> speedups up to **2 orders of magnitude**

- Not just convenient, but a matter of necessity for certain useful computations, like noise studies that require many identical simulation shots

Why GPUs?

- Virtually any numerical package for solving on local machine uses CPU
  - Good for sequential computation, like integrating system coordinates over time

- However, it is possible to use GPUs for parallel computations
  - No speedup for non-parallel operations (can’t add more coordinates/modes)
  - MASSIVE speedup for parallel operations, such as varying a parameter of the system

- Long story short, NVIDIA has a tool called CUDA for accessing super low-level controls of certain GPU models, and a python package makes it possible to write *custom CUDA kernels* to speed up arbitrary equations of motion

NVIDIA CUDA

First try at simulation speedup

- Simple test problem: single-mode system with a driving term and a nonlinearity (Duffing oscillator)

$$\ddot{x} + 2\kappa\dot{x} + \omega_0^2 \sin(x) + \alpha x^3 = 0$$

- We can write this in matrix form by writing

$$\dot{x_1} \equiv x_2$$

$$\dot{x_2} + 2\kappa x_2 + \omega_0^2 \sin(x_1) + x_1^3 = 0$$

- We can add a driving term with

$$\dot{x_1} = x_2 + A\cos(\omega t)$$

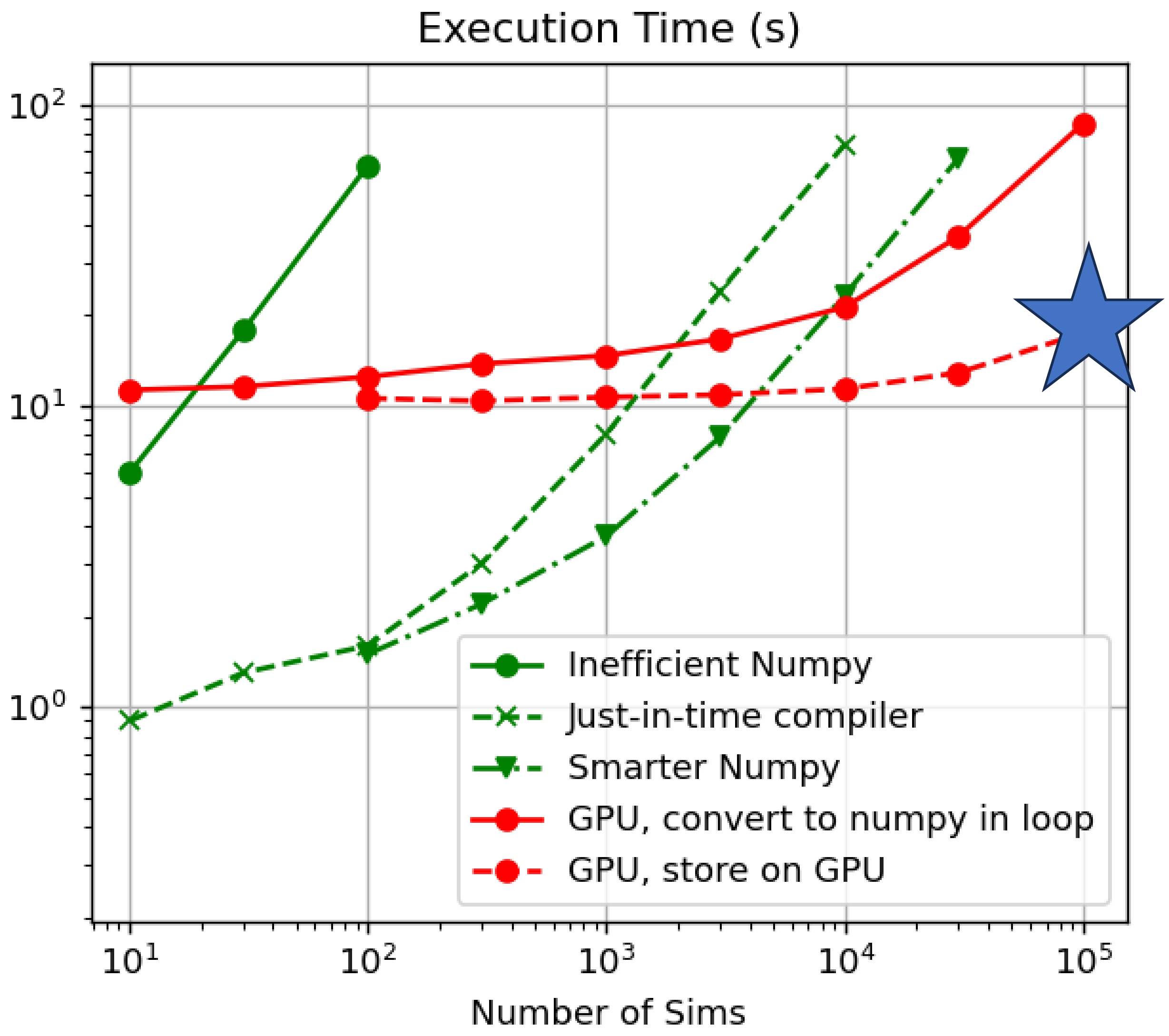
- Or in matrix form:

$$\dot{\mathbf{x}} = \begin{bmatrix} 0 & 1 \\ -\omega_0^2 & -2\beta \end{bmatrix}\mathbf{x} + \begin{bmatrix} 0 & 0 \\ \alpha & 0 \end{bmatrix}\mathbf{x}^3 + \begin{bmatrix} A \\ 0 \end{bmatrix}\cos(\omega_p t)$$

- Implement two Runge-Kutta 4 solvers, one in typical python code, one using custom CUDA kernel for bulk of the calculations

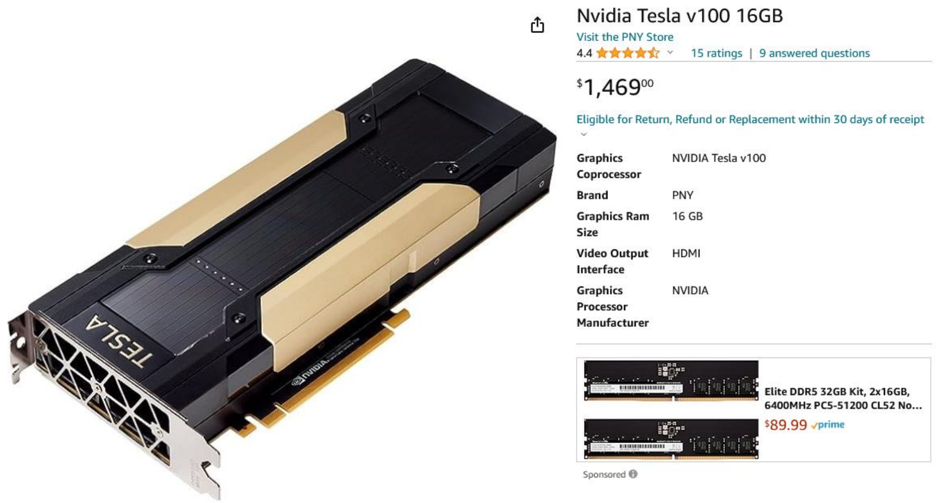
First try results

- Three methods to compare
- All run on Google colab, V100 runtime with high RAM
- 10000 time steps, RK4 integration (40000 function calls)
- Includes some initialization and integration loop (some overhead excluded)
- Slight difference: GPU method stored all trajectories, JIT and vanilla kept only most recent trajectory (so this GPU code uses more RAM)
- About 1.5 orders of magnitude speedup GPU vs JIT
- GPUv2 – faster but uses way more GPU memory (stores trajectories on GPU RAM). Runs out of memory beyond 10^5

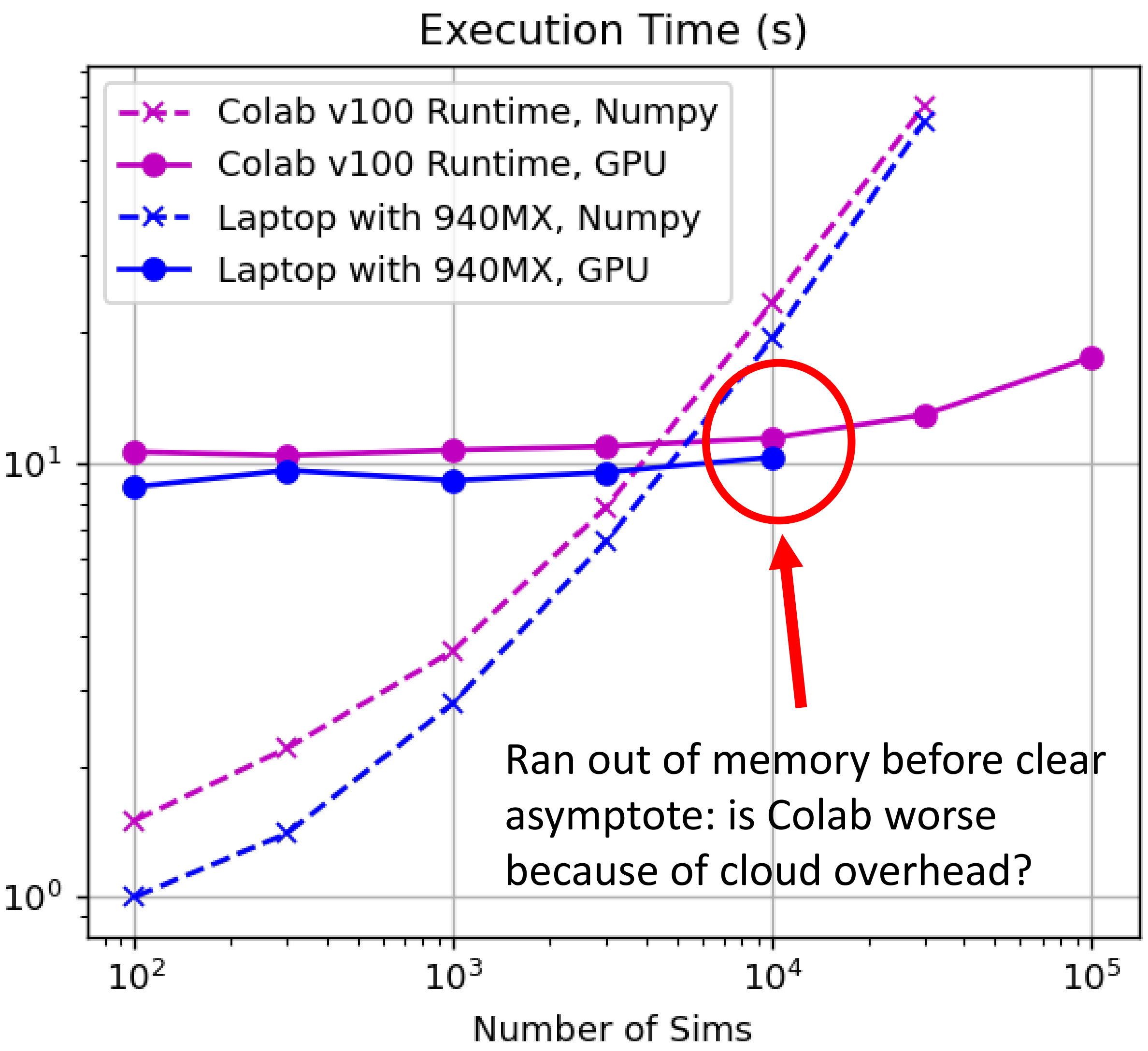


Comparing different GPU hardware

- Google colab has different runtimes, or different hardware you can use
- I used the v100 runtime throughout, which has an NVDA Tesla v100 GPU
  - 16 GB RAM, 5120 CUDA cores



- Compare with my old college laptop with an NVIDIA GeForce 940MX GPU
  - 2 GB RAM, 384 CUDA cores

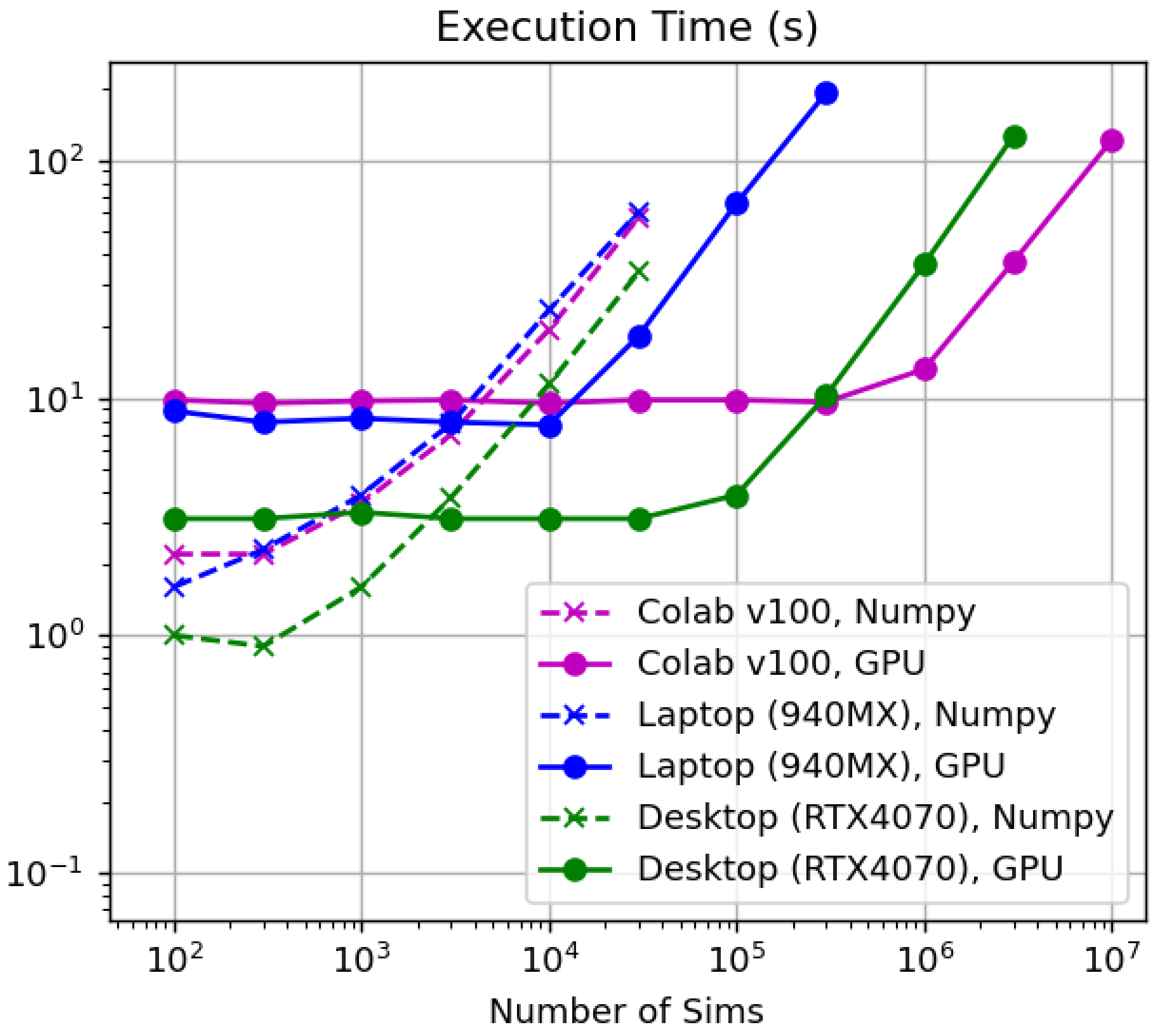


Memory problems

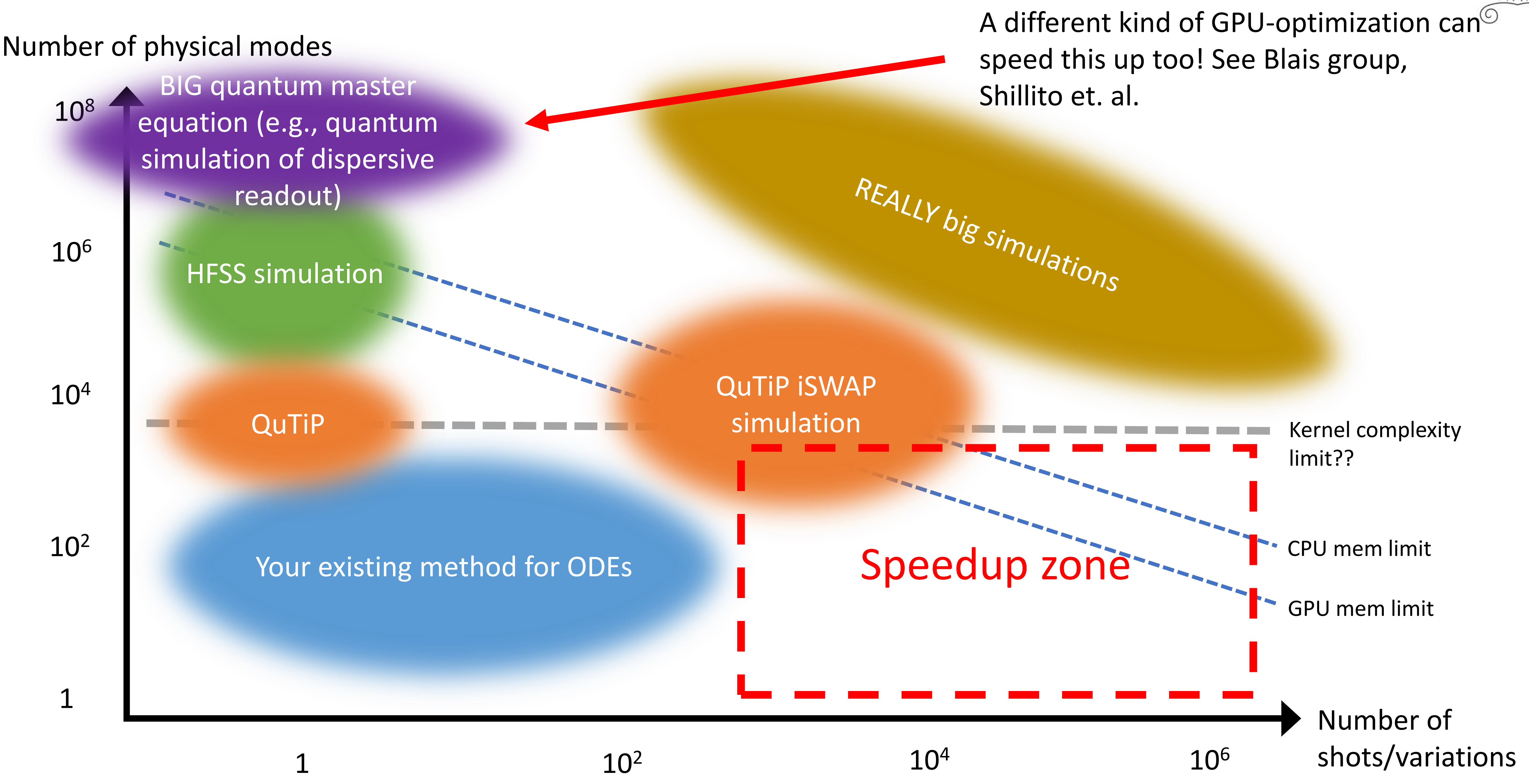
- Ok, big speedups are possible, but we still don't know how big
- Not clear if we are fully utilizing parallelism before we run out of memory (both on CPU and GPU)
- We may be able to do some smart memory management on our own hardware later
  - maybe possible on cloud services?
  - In any case, a problem for another day
- There are classes of simulations that don't have huge memory requirements. E.g.,
  - Spectral power at only 1 frequency
  - Noise characterization
  - Final value
  - Built-in GPU demodulation
- How far can we push these low-memory simulations?

Comparison without GPU memory constraint

- Is colab worse? – Absolutely not, in fact it kicks ass
- Numpy also gets a bit faster without having to save as much data, to be fair
- Notably, a fast GPU lets you *simulate millions of times in less than a minute* – outperforms CPU by at least 2 orders of magnitude
- At around 100 million simulations, each timestep will have 16 GB of data, which is another, harder to circumvent memory limit
- We also bought an RTX 4070, about 1/3 as fast as the v100

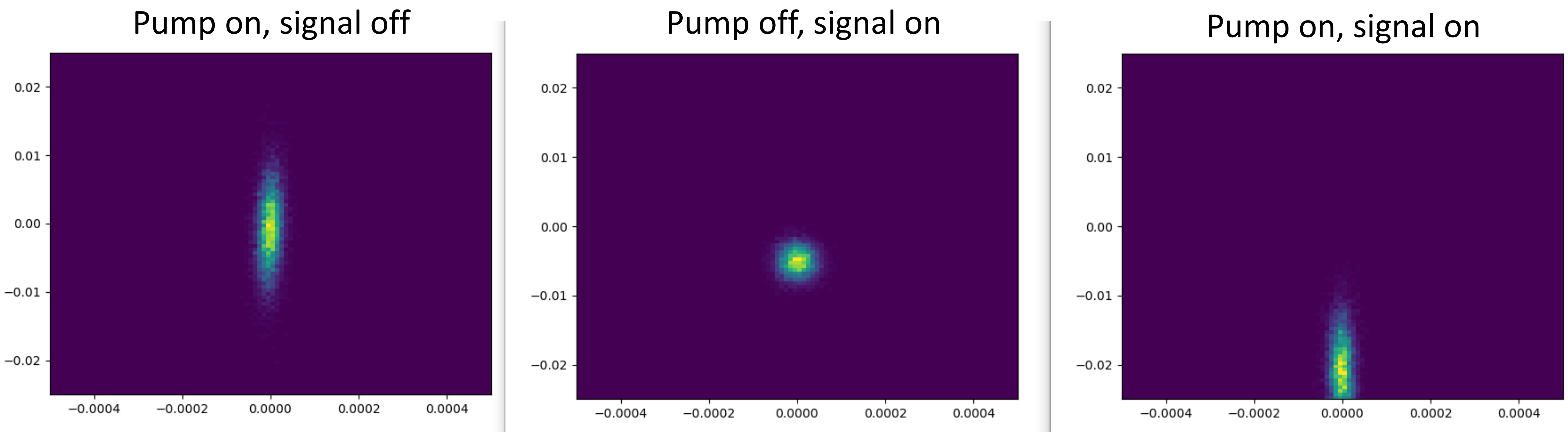


When is this technique useful?



In practice, simple example

- Operating the Duffing oscillator as a single mode degenerate parametric amplifier
- Produce gain on signal tone with pump tone at 3x the frequency, observe final state



- In general, solving equations is not enough – some processing needs to be part of the GPU code
- For example, here we take instantaneous histograms of 10000 identical shots, adding continuous noise to each
- How do we reject pump oscillations from histograms?
- Couple oscillator to additional resonant mode at same frequency, track its dynamics instead (NOT the same as real experiment, which is actually a time-averaging process)

Example code

```
variations1 = 1
variations2 = 10000

var_strs = ['z']
exp_strs = ['1j*z*omega']

params = [('omega', 10 * 2 * np.pi, 20 * 2 * np.pi, 100)]

kernel_input, kernel_output, kernel_body, kernel_op = generate_kernel(var_strs, exp_strs, params,
use_complex=True)

print(kernel_input)
print(kernel_body)
print(kernel_output)

N = len(var_strs)

dt = 0.1/(10 * 2 * np.pi)
steps = 10000
t = np.linspace(0, dt*steps, steps)

start_time = time.time()
x, x_avg = related_rates_problem(t, N, variations1, variations2, kernel_op)
```

Todos

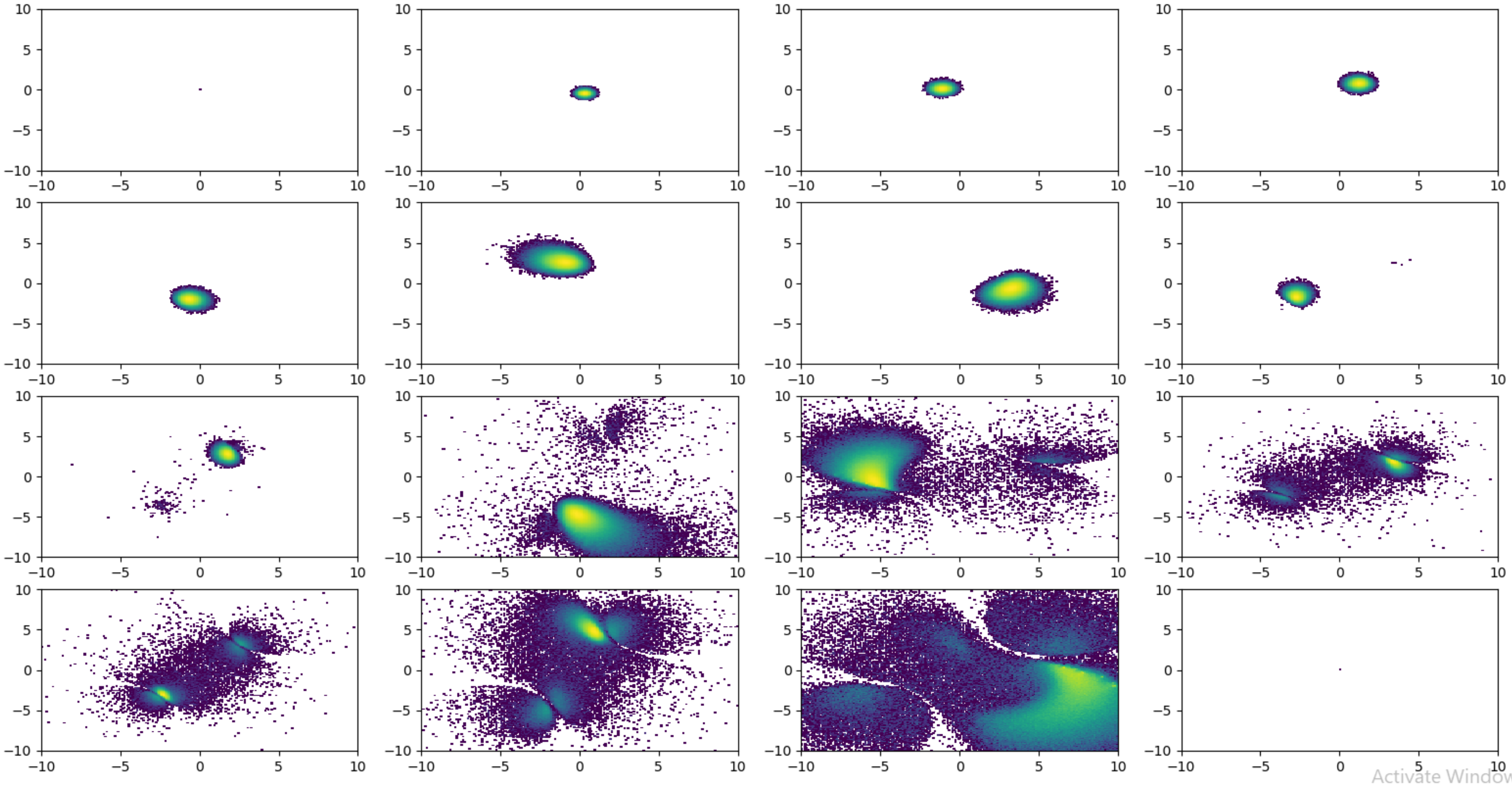
- Done:
  - Benchmarking
  - Solver class
  - Pump reject resonant filtering
- Almost done:
  - GPU demodulation (more realistic processing)
  - Interpret equations of motion as complex (for Langevin equations)
- Would like to do
  - Limit comparison tests
    - In limit of steady state, matches joe theory? Chenxu saturation power?
    - In limit of many modes in a line, matches transmission line theory
  - Apply noise analysis to current amplifier experiments
  - Use on some real problem involving large parameter space

- Caveats: Julia? CUDA Quantum? How much do we actually care?

Extra slides

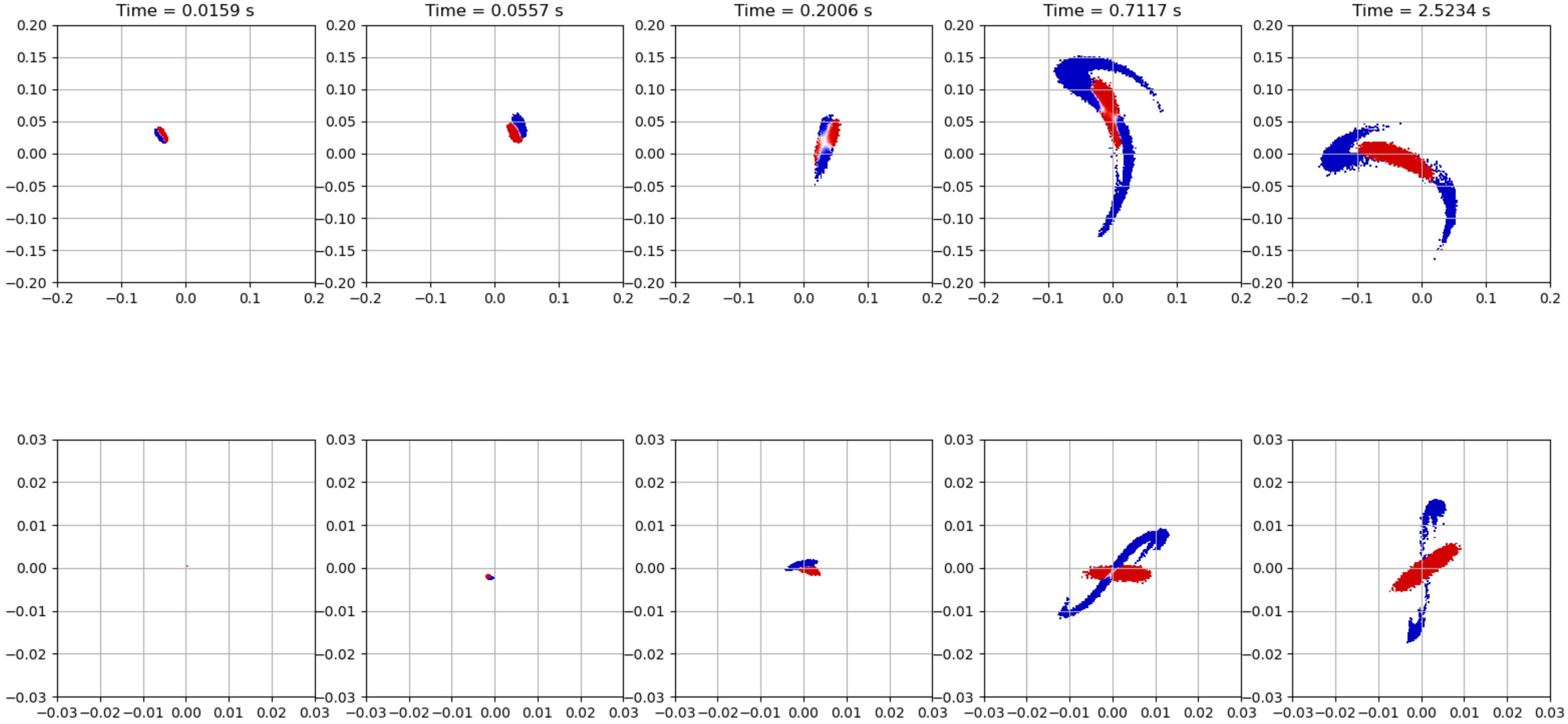
Mostly boring tests confirming things you'd already expect

Example time evolution



Activate Windows
Go to Settings to activate

I don’t know anything

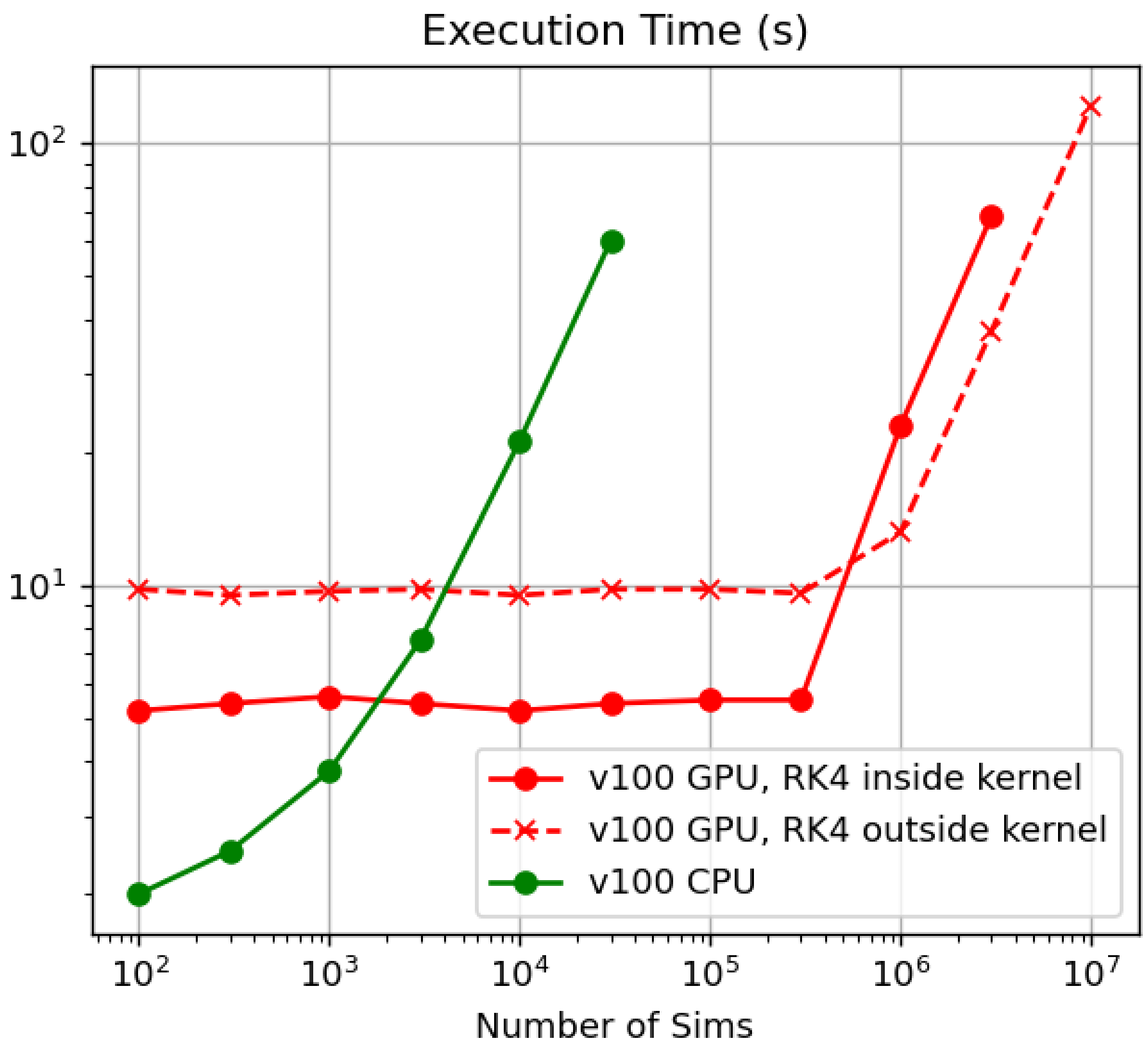


Examples of interesting results

- So far everything has been a basic simulation of a single mode: BORING
- Here are 4 more interesting test cases I would like to examine
  - Amplifier gain bandwidth using decimated data
    - Still single mode, but now do online decimation
  - Chenxu saturation power calculation
    - Includes input-output theory aspects
  - Embedded amplifier final state
    - Three mode, 4 drive
  - Two-mode squeezed light
    - Includes “transmission line” which simulates time-delay, as well as “fake circulator”

What to put in the GPU kernel

- It's not obvious what computations to put in the GPU kernel
- For example, in the RK4 method, each calculation for the next instant in time requires 4 evaluations of the derivative function
- Put only the derivative function in the kernel, and call the kernel function 4 times?
- OR repeat the derivative function in the kernel 4 times, and call the kernel once?
- In my testing, the first seems better, but with more overhead
- Not sure why



Digression about numpy and numba

- Some caveats about the "vanilla" python speed test
  - I think the speed I reported wasn't totally fair to pure python
  - The reason has to do with for loops and a thing called a just-in-time compiler
- Python for loops are slow
  - the computation done in the innermost loop should not be very small, since execution time will be dominated by just calling the for loop
  - put as much non-redundant computation as possible in innermost loop
  - one way is to use numpy indexing
- Numpy matrix math is faster than manually doing it in python
- The only time numba is necessarily better than pure numpy is when you want to implement some weird or arbitrary matrix logic that isn't found in a numpy function, and that probably won't happen in our ODEs. So numba is purely for convenience

```python
@njit
def ident_np(x):
    return np.cos(x) ** 2 + np.sin(x) ** 2

@njit
def ident_loops(x):
    r = np.empty_like(x)
    n = len(x)
    for i in range(n):
        r[i] = np.cos(x[i]) ** 2 + np.sin(x[i]) ** 2
    return r
```

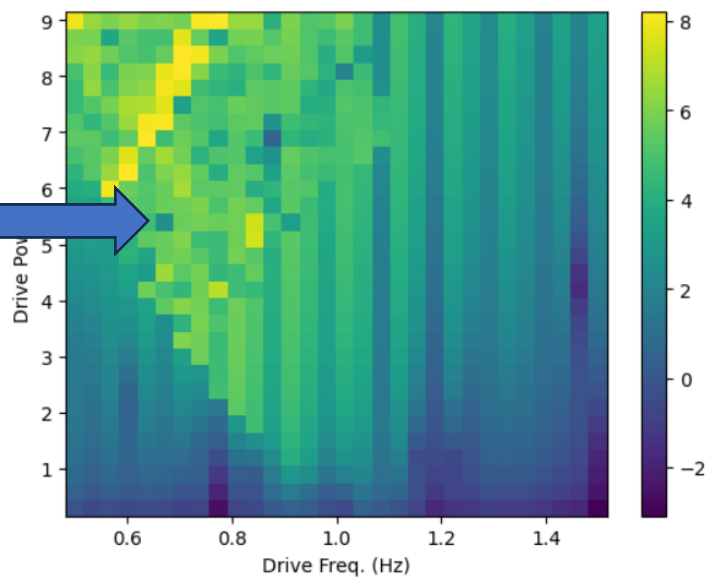
Function Name	@njit	Execution time
`ident_np`	No	0.581s
`ident_np`	Yes	0.659s
`ident_loops`	No	25.2s
`ident_loops`	Yes	0.670s

Why ever use JIT?

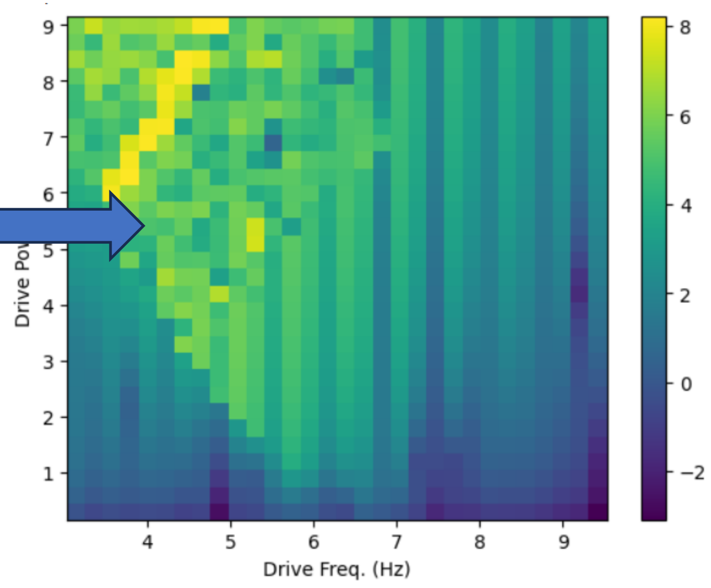
- The reason we might use JIT is if we have some logic in our calculations that cannot be represented with array operations
- As a silly example, for each matrix element, if the row modulo the column is a prime number, add a number that increments by 1 each time to that element
- This would be hard (not possible?) with numpy operations like + or np.dot()
- Then you could write a for loop and speed it up with JIT
- These kind of situations might arise when designing pulse sequences, I would guess

Results difference: floating point precision?

Example calculation for magnitude of third-order harmonic in a driven damped nonlinear oscillator



JIT



GPU

What's the computational difference between variations and shots?

- In my terminology, shots are identical copies of the simulation, and variations are copies with different values of parameters
- Either way, the simulation runs in parallel, so is there any reason why, for example, 1 million shots of 1 variation should run slower/faster than 1 million variations with 1 shot each? What about a thousand variations with a thousand shots each?
- No difference (I tested exactly this case on both GPU and CPU)

A tiny bit of memory management

- Basically, if you're storing all the traces on the GPU RAM, just call this after each batch of simulations:

```
del(x)
  cp._default_memory_pool.free_all_blocks()
```

Original use: parallel processing for fractal generation ☺

Convergence map for Newton-Raphson root finding algorithm on the complex function `'z**3 + cos(z**2) - 1 + 3j'`

3200x3200 image in 3 seconds