WrightSim

Multidimensional Spectroscopy Simulation using Numerical Integration

Kyle Sunden

December 21, 2017

Abstract

Presented here is a reimplementation of a package for simulating quantum mechanical non-linear spectra. This package uses numerical integration to evolve a density state vector over time as the system interacts with several electric fields. Herein, an existing implementation, `NISE`, is analyzed, and algorithmic improvements made for a new implementation, `WrightSim`. The effect of parallelizing the simulation, both at the CPU and GPU (CUDA) levels, is demonstrated. This simulation is highly paralellizable, and benefits by multiple orders of magnitude from the combined effects of algorithmic improvements and parallelism. Many possible further improvements are also discussed, such as improving the memory footprint of the simulations. `WrightSim` has taken a huge leap forward towards the goal of being a fast, easy to use, experimentalist friendly simulation package.

CONTENTS

1 Theory and Background

The information presented in this section is largely a reproduction, edited for brevity, of that which was presented by Kohler, Thompson, and Wright[1]. Readers interested in finer grained detail are encouraged to reference their work, in particular the supplemental information. Below, I discuss the broader goals of this project, provide an insight into the mathematics underlying the simulation, and discuss an existing implementation of a software package to perform the simulations.

1.1 Goals

The primary goal of `WrightSim` is to reproduce *in silico* the multidimensional spectra obtained by experimentalists like those found within the Wright Group. `WrightSim` is designed with the experimentalist in mind, allowing users to parameterize their simulations in much the same way that they would collect a similar spectrum in the laboratory. Numerical integration is used to provide the most flexible, accurate, and interpretable simulations possible. While the focus is on frequency domain nonlinear spectroscopy, in principle the technique is applicable to time domain approaches as well. There are other levels of theory which can provide more computationally efficient analytical solutions, however these make assumptions which are not always valid.
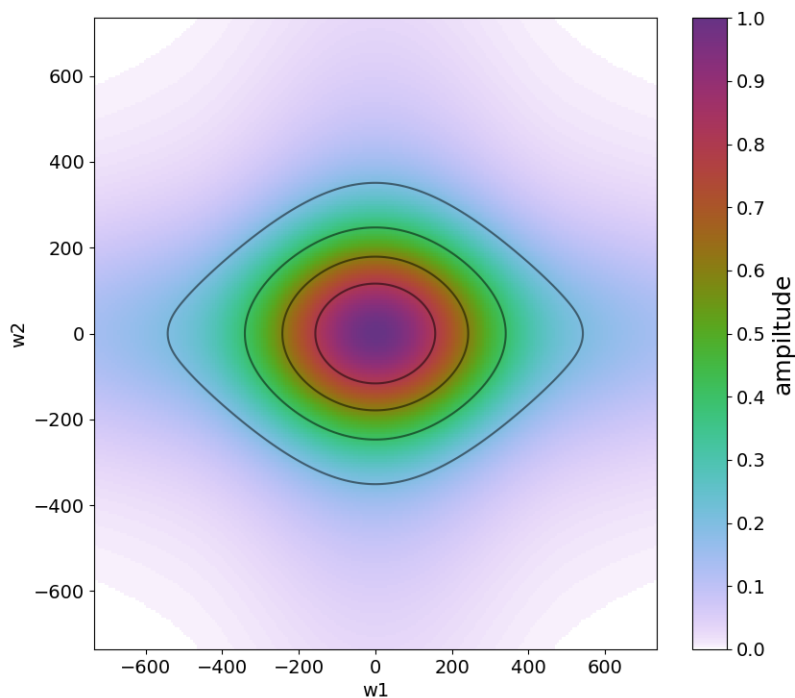

Figure 1.1: Simulated spectrum at normalized coordinates

Figure 1.1 shows an example visualization of a spectrum in 2-Dimensional frequency space. The axes are two different frequencies for the input electric fields, the axes are normalized such that there is a resonance around 0.0 in both frequencies. While this particular visualization took a more naïve approach, encoded within the values available are information about the frequency and phase of the output signal. This enables users to select a particular portion of the spectrum to visualize, similar to using a monochromator in the laboratory. This technique allows for traversals through many aspects of experimental space. Shown is in frequency space, however we also have control over temporal separation of pulses, and can scan against this delay space. Another advantage of this package, is that it allows users to be selective of which portions of the signal are simulated, providing a deeper understanding of how the shape of a spectrum arises from the underlying quantum mechanics.

1.2 A BRIEF INTRODUCTION ON RELEVANT QUANTUM MECHANICS

If you are interested in a more full understanding, please refer to Kohler, Thompson, and Wright[1]. This introduction is intended to very quickly introduce *what* is being done, but not *why*.

Here, we are simulating the interactions of three electric fields to induce an output electric field. These fields can interact in a combinatorially large number of fashions. For three fields, there are six possible time orderings for the pulses to interact and create superpositions or populations in the material system. We are restricting this simulation to have two positive interactions (solid up arrows or dashed down arrows) and one negative interaction (dashed up arrow or solid down arrow). This results in 16 independently resolvable possible pathways which result in a productive emission. Figure 1.2 shows these 16 pathways, arranged by their time ordering (I - VI). For the purposes of this paper it is not necessary to fully understand what is meant by this diagram, the intent is simply to show the complexity of the problem at hand. Experimentalists can isolate the time orderings by introducing delays between pulses to preferentially follow the according pathways. Simulation allows us to fully separate these time orderings and pathways, to as fine a detail as desired.

Figure 1.2 shows a finite state automata, starting at the ground state (ρ_{00}). Encoded within each node is both the quantum mechanical state and the fields with which the system has already interacted. Interactions occur along the arrows, which transfer density from one state to another. In order to fit the parameters of this simulation, the fields must each interact exactly once. Output is generated by the rightmost two nodes, which have interacted with all three fields. These nine states represent all possible states which match the criterion described by the process we are simulating. We can take these nine states and collect them into a state density vector, $\overline{\rho}$ (Equation 1.1).

$$\overline{\rho} \equiv \begin{bmatrix} \tilde{\rho}_{00} \\ \tilde{\rho}_{01}^{(-2)} \\ \tilde{\rho}_{10}^{(2')} \\ \tilde{\rho}_{10}^{(1)} \\ \tilde{\rho}_{20}^{(1+2')} \\ \tilde{\rho}_{11}^{(1-2)} \\ \tilde{\rho}_{11}^{(2'-2)} \\ \tilde{\rho}_{10}^{(1-2+2')} \\ \tilde{\rho}_{21}^{(1-2+2')} \end{bmatrix} \tag{1.1}$$

Next we need to quantitate the transitions between states. This is the Hamiltonian matrix. Since we have nine states in our density vector, the Hamiltonian is a nine by nine matrix. To assist in representing the matrix, six time dependent variables are defined:

$$A_1 \equiv \frac{i}{2}\mu_{10}e^{-i\omega_1\tau_1}c_1(t-\tau_1)e^{i(\omega_1-\omega_{10})t} \tag{1.2}$$

$$A_2 \equiv \frac{i}{2}\mu_{10}e^{i\omega_2\tau_2}c_2(t-\tau_2)e^{-i(\omega_2-\omega_{10})t} \tag{1.3}$$

$$A_{2'} \equiv \frac{i}{2}\mu_{10}e^{-i\omega_{2'}\tau_{2'}}c_{2'}(t-\tau_{2'})e^{i(\omega_{2'}-\omega_{10})t} \tag{1.4}$$

$$B_1 \equiv \frac{i}{2}\mu_{21}e^{-i\omega_1\tau_1}c_1(t-\tau_1)e^{i(\omega_1-\omega_{21})t} \tag{1.5}$$

$$B_2 \equiv \frac{i}{2}\mu_{21}e^{i\omega_2\tau_2}c_2(t-\tau_2)e^{-i(\omega_2-\omega_{21})t} \tag{1.6}$$

$$B_{2'} \equiv \frac{i}{2}\mu_{21}e^{-i\omega_{2'}\tau_{2'}}c_{2'}(t-\tau_{2'})e^{i(\omega_{2'}-\omega_{21})t} \tag{1.7}$$
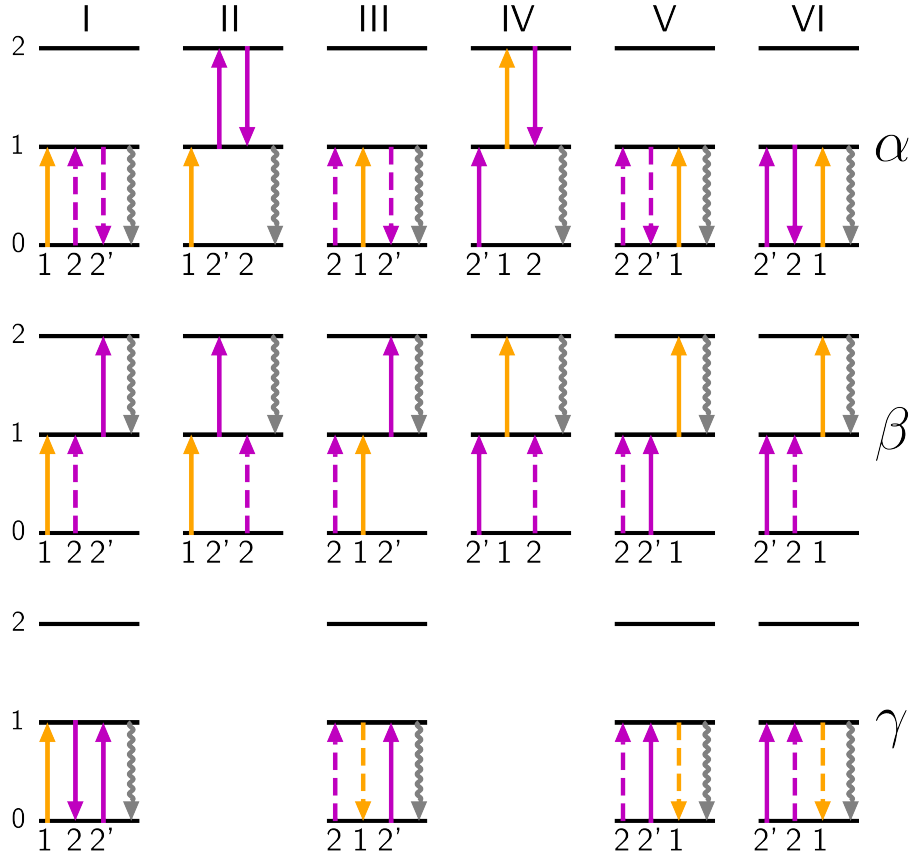

Figure 1.2: Independent Lioville pathways simulated. Excitations from ω_1 are in yellow, excitations from $\omega_2 = \omega_{2'}$ are shown in purple. Figure was originally published as Figure 1 of Kohler, Thompson, and Wright[1]
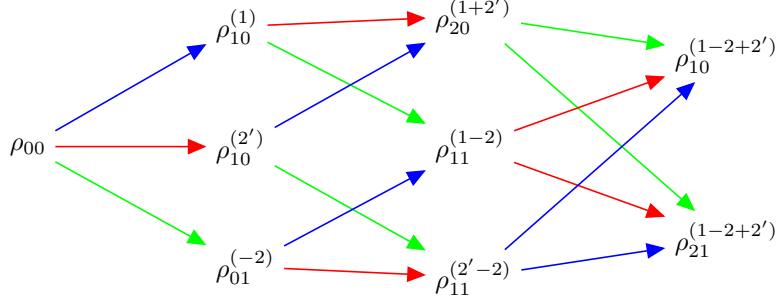
Figure 1.3: Finite state automata of the interactions with the density matrix elements. Matrix elements are denoted by their coherence/population state (the subscript) and the pulses which they have already interacted with (the superscript). Arrows indicate interactions with ω_1 (blue), $\omega_{2'}$ (red), and ω_2 (green). Figure was originally published as Figure S1 of Kohler, Thompson, and Wright[1]

These variables each consist of a constant factor of $\frac{i}{2}$, a magnetic susceptibility term ($\mu_{10|21}$), an electric field phase and amplitude (the first exponential term), an envelope function (c, a Gaussian function here), and a final exponential term. These variables can then be used to populate the matrix:

$$
\overline{\overline{Q}} \equiv
\begin{bmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-A_2 & -\Gamma_{10} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
A_{2'} & 0 & -\Gamma_{10} & 0 & 0 & 0 & 0 & 0 & 0 \\
A_1 & 0 & 0 & -\Gamma_{10} & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & B_1 & B_{2'} & -\Gamma_{20} & 0 & 0 & 0 & 0 \\
0 & A_1 & 0 & -A_2 & 0 & -\Gamma_{11} & 0 & 0 & 0 \\
0 & A_{2'} & -A_2 & 0 & 0 & 0 & -\Gamma_{11} & 0 & 0 \\
0 & 0 & 0 & 0 & B_2 & -2A_{2'} & -2A_1 & -\Gamma_{10} & 0 \\
0 & 0 & 0 & 0 & -A_2 & B_{2'} & B_1 & 0 & -\Gamma_{21}
\end{bmatrix}
\tag{1.8}
$$

The Γ values along the diagonal represent dephasing, that is, loss of coherence, which happens without any interaction. To isolate a given time ordering, we can simply set the value of elements which do not correspond to that time ordering to zero.

At each time step, the dot product of the matrix with the $\overline{\rho}$ vector is the change in the $\overline{\rho}$ vector to the next time step (when multiplied by the differential). Both `NISE` and `WrightSim` use a more advanced, second order technique (Runge-Kutta) for determining the change in the $\overline{\rho}$ vector, but the idea is the same. The core of the simulations is to take the $\overline{\rho}$ vector and multiply by the Hamiltonian at each time step (noting that the Hamiltonian is time dependant, as are the electric fields). This process repeats over a large number of small time steps, and must be performed separately for any change in the inputs (e.g. frequency [ω] or delay[τ]).

1.3 NISE: NUMERICAL INTEGRATION OF THE SHRÖDINGER EQUATION

`NISE` [2] is the open-source package written by Kohler and Thompson while preparing their manuscript [1]. `NISE` uses a slight variation on the technique described above, whereby they place a restriction on the time ordering represented by the matrix, and can thus use a seven element state vector rather than a 9 element state vector. This approach is mathematically equivalent to that presented above. The approach presented is what is used in `WrightSim`. The trade off is that to obtain a full picture, they must build in a

mechanism to perform two simulations at the same time, increasing complexity, and actually reducing performance.

NISE is included here as a reference for the performance of previous simulations of this kind.

2 ALGORITHMIC IMPROVEMENTS

When first translating the code from NISE into the paradigm of WrightSim, I sought to understand why it took so long to compute. I used Python's standard library package cProfile to produce traces of execution, and visualized them with SnakeViz[3]. Figure 2.1 shows the trace obtained from a single-threaded run of NISE simulating a $32x32x16$ frequency-frequency-delay space. This trace provided some interesting insights into how the algorithm could be improved. First, 99.5% of the time is spent inside of a loop which is highly parallelizable. Second, almost one third of that time was spent in a specific function of numpy, ix_. Further inspection of the code revealed that this function was called in the very inner most loop, but always had the same, small number of parameters. Lastly, approximately one tenth of the time was spent in a particular function called rotor (the bright orange box in Figure 2.1). This function computed $cos(theta) + i * sin(theta)$, which could be replaced by the equivalent, but more efficient $exp(i * theta)$. Additional careful analysis of the code revealed that redundant computations were being performed when generating matrices, which could be stored as variables and reused.

When implementing WrightSim, I took into account all of these insights. I simplified the code for matrix generation and propagation by only having the one 9 by 9 element matrix rather than two 7 by 7 matrices. The function that took up almost one third the time (ix_) was removed in favor of a simpler scheme for denoting which values to record. I used variables to store the values needed for matrix generation, rather than recalculating each element. As a result, solely by algorithmic improvements, almost an order of magnitude speedup was obtained (See Figure 2.2). Still, 99% of the time was spent within a highly parallelizable inner loop.
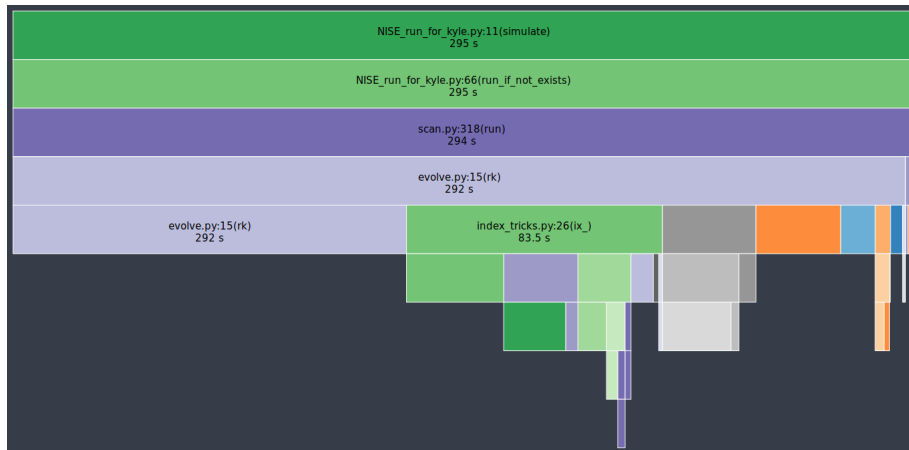

Figure 2.1: Profile trace of a single threaded simulation from NISE.

3 CPU AND GPU PARALLEL IMPLEMENTATIONS

NISE already had, and WrightSim inherited, CPU multiprocessed parallelism using the Python standard library multiprocessing interface. Since almost all of the program is parallelizable, this incurs a four times
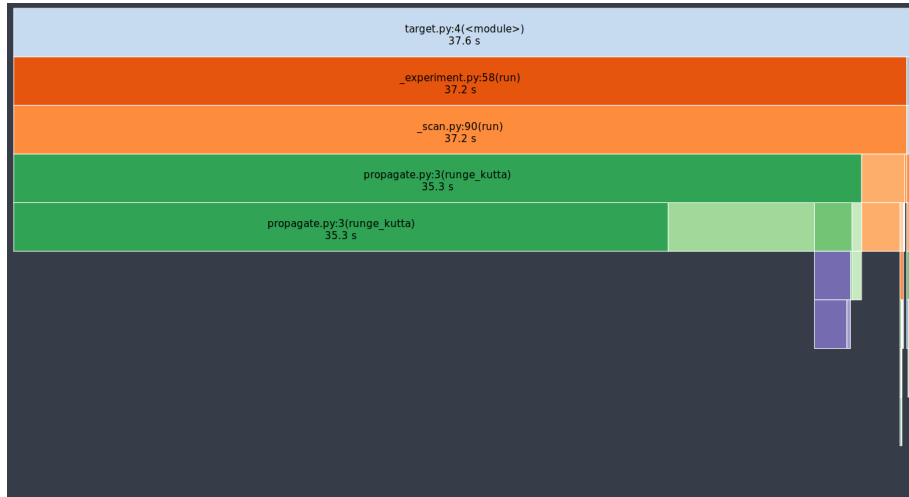
Figure 2.2: Profile trace of a single threaded simulation from `WrightSim`.

speedup on a machine with four processing cores (limited more by the operating system scheduling other tasks than by Amdahl's law). This implementation was not adjusted outside of minor API tweaks.

In order to capitalize as much as possible on the amount of parallelism possible, an implementation using Nvidia CUDA [4] was performed. In order to make the implementation as easy to use as possible, and maintainable over the lifetime of `WrightSim`, PyCUDA was used to integrate the call from within python to a CUDA kernel. `PyCUDA` allows the source code for the device side functions (written in C/C++) to exist as strings within the python source files. These strings are just in time compiled (using `nvcc`) immediately prior to calling the kernel. For the initial work with the CUDA implementation, only one Hamiltonian and one propagation function were written, however it is extensible to additional methods. The just in time compilation makes it easy to replace individual functions as needed (a simple form of metaprogramming).

The CUDA implementation is slightly different from the pure Python implementation. It only holds in memory the Hamiltonian matrices for the current and next step, where the Python implementation computes all of the Matrices prior to entering the loop. This was done to conserve memory on the GPU. Similarly, the electric fields are computed in the loop, rather than computing all ahead of time. These two optimizations reduce the memory overhead, and allow for easier to write functions, without the help of numpy do perform automatic broadcasting of shapes.

3.1 SCALING ANALYSIS

Scaling analysis, testing the amount of time taken by each simulation versus the number of points simulated, were conducted for each of the following: `NISE` single threaded, `NISE` Multiprocessed using four cores, `WrightSim` Single threaded, `WrightSim` Multiprocessed using four cores, and `WrightSim` CUDA implementation. A machine with an Intel Core i5-7600 (3.5 GHz) CPU and an Nvidia GTX 1070 graphics card, running Arch Linux was used for all tests. The simulations all had the same end goal for simulation, with the same number of time steps and same recorded values. The `NISE` simulations use two seven by seven matrices for the Hamiltonian, while the `WrightSim` simulations use a single nine by nine matrix. The results are summarized in Figure 3.1.

The log-log plot shows that the time scales linearly with number of points. All lines have approximately the same slope at high values of N, though the CUDA implementation grows slower at low N. The Algorithmic improvements alone offer doubled performance over even 4-Core multiprocessed `NISE`
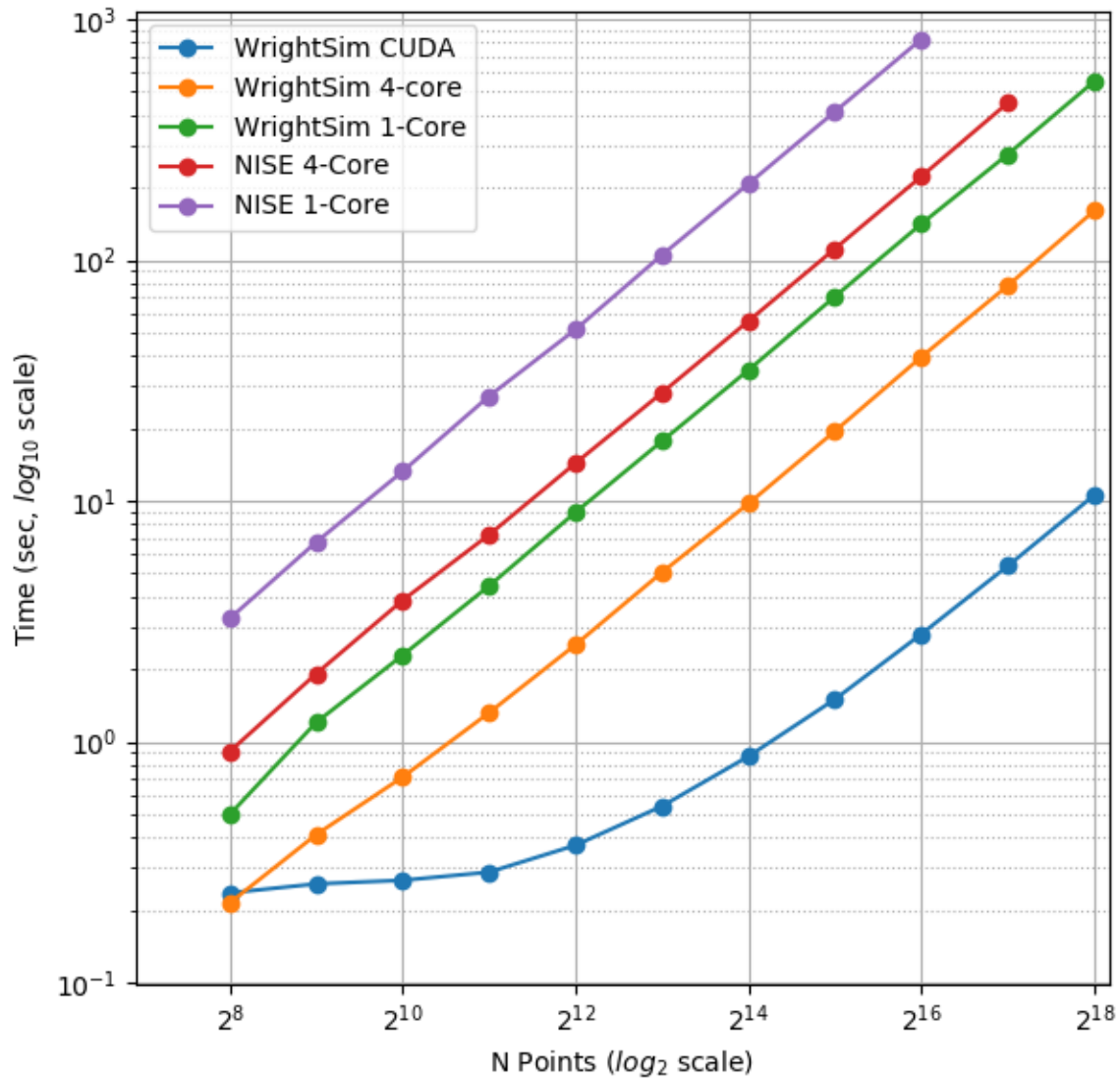
Figure 3.1: Scaling Comparison of `WrightSim` and `NISE`

simulation. The CUDA implementation has a positive intercept at approximately 200 milliseconds. This is due, in large part, to the compilation overhead.

3.2 LIMITATIONS

The CUDA implementation faces limitations at both ends in terms of number of points. On the low side, the cost of compilation and transfer of data makes it slower than the 4-Core CPU Multiprocessing implementation. This crossover point is approximately 256 points (for this simulation, all other parameters

being equal). Incidentally, that is also a hard coded block size for the CUDA kernel call. While this could be modified to ensure no illegal memory accesses occur on smaller cases, the fact that you are not saving by using CUDA (and even single core performance is under a second) means it is not worth the effort at this time. The hard-coded block size also means that multiples of 256 points must be used in the current implementation.

On the high number of points side, we are limited by the amount of memory allowed to be allocated on the GPU. For each pixel in the simulations presented here, 250 complex numbers represented as doubles must be allocated. Additional data is needed, but, especially at the high end, it is dominated by this array, which contains the outputs which are then transferred back to the host. Each CUDA thread additionally dynamically allocates the arrays it needs to perform the computation. The current implementation has a limit somewhere between 2^{18} and 2^{19} points. This limit could be increased by using single precision floating point numbers to represent the complex arrays, if the trade-off is acceptable.

4 FUTURE WORK

This is still quite early days for `WrightSim`. While it is already a startling proof of concept display of how High Performance Computing techniques can be applied to this problem, there is still much room for improvement. In general, there are improvements to be made in terms of features, API and ease of use, and indeed further algorithmic improvements.

4.1 FEATURES

`NISE` had implemented a few additional features which were not carried over to `WrightSim` during the development efforts which focused on performance thus far.

There was support for chirped electric field pulses, which behave in less ideal fashions than the true sinusoids and Gaussian peaks used thus far. These non-ideal perturbations can have a real effect in spectra collected in the lab, and accurately modelling them helps to interpret these spectra.

Samples in laboratory experiments may have some amount of inhomogeneity within the sample, resulting in broader than would otherwise be expected peaks. This inhomogeneity can be modeled by storing the response array which is calculated by numerical integration, and translating the points slightly. The original `NISE` implementation would perform the simulation multiple times, where that is not needed as a simple translation will do. At one point we considered generating a library of responses in well known coordinates and saving them for future use, avoiding the expensive calculation all together. That seems to be less needed, given the speed of the CUDA code.

`NISE` provided a powerful and flexible set of tools to "Measure" the signal, using Fourier transforms and produce arrays that even further mimic what is observed experimentally. That system needs to be added to `WrightSim` for it to be feature-complete. More naïve methods of visualizing work in this case, but a true measurement would be better.

Some new features could be added, including saving intermediate responses using an HDF5 based file format. The CUDA implementation itself would benefit from some way of saving the compiled code for multiple runs, removing the 0.2 second overhead. Current implementation compiles directly before calling the kernel, whether it has compiled it before or not. If performing many simulation in quick succession with the same C code, the savings would add up.

The just in time compilation enables some fancy metaprogramming techniques which could be explored. The simple case is using separately programmed functions which have the same signature to do tasks in different ways. Currently there is a small shortcut in the propagation function which uses statically allocated arrays and pointers to those arrays rather than using dynamically allocated arrays. This relies on knowing the size at compilation time. The numbers could be replaced by preprocessor

macros which are also fed to the compiler to assign this value dynamically at compilation time. A much more advanced metaprogramming technique could, theoretically, generate the C struct and Hamiltonian generation function by inspecting the python code and performing a translation. Such a technique would mean that new Hamiltonians would only have to be implemented once, in Python, and users who do not know C would be able to run CUDA code.

4.2 Usability

One of the primary reasons for reimplementing the simulation package is to really think about how users interact with the package. As much as possible, the end user shouldn't need to be an experienced programmer to be able to get a simulation. One of the next steps for `WrightSim` is to take a step back and ensure that our API is sensible and easy to follow. We wish to, as much as possible, provide ways of communicating through configuration files, rather than code. Ultimately, a GUI front end may be desirable, especially as the target audience is experimentalists.

Additional Hamiltonians would make the package significantly more valuable as well. To add more Hamiltonians will require ensuring the code is robust, that values are transferred as expected. A few small assumptions were made in the interest of efficiency in the original implementation. Certain values represented by the Hamiltonian were hard-coded on the device code Those changes should be reversed.

4.3 Further Algorithmic Improvements

While great strides were taken in improving the algorithms from previous implementations, there are several remaining avenues to gain even further. The CUDA implementation is memory bound, both in terms of what can be dispatched, and in terms of time of execution. The use of single precision complex numbers (and other floating point values) would save roughly half of the space. One of the inputs is a large array with parameters for the each electric field at each pixel. This array contains much redundant data, which could be compressed with the parsing done in parallel on the device.

If the computed values could be streamed out of the GPU once computed, while others use the freed space, then there would be almost no limit on the number of points. This relies on the ability to stream data back while computation is still going, which I do not have experience doing, and am not sure CUDA even supports. The values are not needed once they are recorded, so there is no need from the device side to keep the values around until computation is complete.

Additional memory could be conserved by using a bit field instead of an array of chars for determining which time orderings are used as a boolean array. This is relatively minimal, but is a current waste of bits. The Python implementation could potentially see a slight performance bump from using a boolean array rather than doing list searches for this same purpose.

The CUDA implementation does not currently take full advantage shared cache. Most of the data needed is completely separated, but there are still a few areas where it could be useful. The Hamiltonian itself is shared, and if the electric field parameters array is sent in a more compressed format, it would be shared as well.

The current CUDA implementation fills the Hamiltonian with zeros at every time step. The values which are nonzero after the first call are always going to be overwritten anyway, so this wastes time inside of of nested loop. This zeroing could be done only before the first call, removing a nested loop. Many matrices have a lot of zero values. Often they are triangular matrices, which would allow for a more optimized dot product computation which ignores the zeros in the half which is not populated. Some matrices could even benefit by being represented as sparse matrices, though these are more difficult to use.

Finally, perhaps the biggest, but also most challenging, remaining possible improvement would be to capitalize on the larger symmetries of the system. It's a non-trivial task to know which axes are symmetric,

but if it could be done, the amount that actually needs to be simulated would be much smaller. Take the simulation in Figure 1.1. This was computed as it is displayed, but there are two orthogonal axes of symmetry, which would cut the amount actually needed to replicate the spectrum down by a factor of four. Higher dimensional scans with similar symmetries would benefit even more.

5 CONCLUSIONS

`WrightSim`, as implemented today, represents the first major step towards a cohesive, easy to use, fast simulation suite for quantum mechanical numerically integrated simulations. Solely algorithmic improvements enabled the pure python implementation to be an order of magnitude faster than the previous implementation. The algorithm is highly parallelizable, enabling easy CPU level parallelism. A new implementation provides further improvement than the CPU parallel code, taking advantage of the GP-GPU CUDA library. This implementation provides approximately 2.5 orders of magnitude improvement over the existing `NISE` serial implementation. There are still ways that this code can be improved, both in performance and functionality, but it is a truly amazing start to this project.

6 ACKNOWLEDGMENTS

I would like to thank Blaise Thompson and Dan Kohler for their assistance in getting this project started. They provided a lot of help in understanding both the theory from their paper and the code they wrote in `NISE`. They ported many of the ancillary bits of `NISE` to help get something functional which could then be improved. They provided test cases and interpretation, helping to track down bugs and brainstorm ideas.

This project is truly the vision of Blaise Thompson, it would not be extant without him. He has many plans for the future directions for this project, and I look forward to implementing them and contributing my own.

References

[1] Kohler, D. D.; Thompson, B. J.; Wright, J. C. *The Journal of Chemical Physics* **2017**, *147*, 084202.

[2] Group, W. NISE: Numerical Integration of the Shrödinger Equation. 2016; `http://github.com/wright-group/NISE`.

[3] jiffyclub, SnakeViz. 2017; `http://jiffyclub.github.io/snakeviz/`.

[4] Nickolls, J.; Buck, I.; Garland, M.; Skadron, K. *Queue* **2008**, *6*, 40.