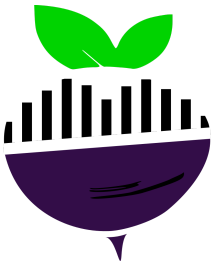
Pankaj Mishra

pankajmishra1511@gmail.com
github.com/pkj-m
+91 831-786-5280

Accelerate Synthetic Spectra Calculations



Mentors: Erwan Pannier *@erwanp*, Dirk van den Bekerom *@dcmvdbekerom*, Nicolas Minesi *@minouHub*

Google Summer of Code 2020 | OpenAstronomy

Overview

RADIS [1] is a fast line-by-line code for synthesizing and fitting infrared absorption and emission spectra such as encountered in laboratory plasmas or exoplanet atmospheres.

It uses the line-by-line approach, where each absorption or emission line of a molecule is computed by accounting for the influence of environment parameters, which change the shape of an absorption or emission line from a Dirac to a convolution of a Gaussian and a Lorentzian profile (called a Voigt profile). Although this method is the only way to calculate highly accurate spectra, it requires computation of a convolution for each individual line. In high-temperature cases (combustion, plasmas, exoplanets), spectra may contain dozens of millions of lines which makes the calculation extremely computationally intensive, and may take up to multiple hours per spectrum.

RADIS introduced a new approach [2] to compute the lineshape broadening step. Recent developments already made the code among the fastest in the world, compared to equivalent codes at NASA [3] and JAXA [4], which is highly relevant for the synthesis of high bandwidth, high resolution spectra. In contrast to the conventional approach, this new approach is based on simple linear algebra operations and can take advantage of hardware-parallelization, which is not implemented in the current version. Proof-of-principle code demonstrates that by unloading the computational intensive part to a GPU, performance increase of 100 000x compared to the state-of-the art may be obtained. The aim of this project is to implement- and optimize the proof-of-principle code into RADIS to take full advantage of the parallel processing power of the GPU.

Goals

The goal of this project is to parallelize the bottleneck operation of synthetic spectra calculations in RADIS, i.e., the lineshape broadening step.

The lineshape broadening calculation currently has multiple variants and pathways that depend on the input parameters, and thus a major task of this project will be to make the GPU-implementation

compliant with all the variants. In the process of restructuring the algorithms and parallelizing them, the entire broadening process will require to be refactored.

By the end of the coding period, I hope to accomplish the following objectives:

1. All variants of the lineshape broadening step are made GPU compatible
2. Parts of the code rendered useless by the new modifications removed and the code base refactored
3. Proof of concept for line-of-sight spectra technique will be implemented.
4. Prepare an ipython notebook containing examples and documentation to allow users to become familiar with the new features

To proceed, we will first start from a minimal line-by-line code and implement the GPU parallelization. Then, we will work with the production code.

In the following section, I will clarify the implementation details and underlying physics of the RADIS spectral code, to get familiar with the code architecture and get started with the project. In Section 3 I'll detail how I plan to implement the new features.

Project Description

Physical Background

Broadening refers to the broadening of the lines in the spectrum due to the interaction of the molecule with its environment. There are two main distinct broadening mechanisms. The first is the result of the different velocities of the emitting particles causing Doppler shifts, the cumulative effect of which is Gaussian line broadening. This is known as Doppler Broadening. Similarly, it is possible for the spectrum lines to broaden due to the collisions of gas molecules with the emmeter, which leads to Pressure Broadening (or Collisional Broadening), and has the shape of a Lorentzian profile. The Voigt Profile is a convolution of Gaussian and Lorentzian profiles and corresponds to what is observed in experiments. The computation of Voigt profiles is non-trivial to optimize and ends up becoming the bottleneck of the spectrum calculation.

Literature optimization techniques

There exist many techniques which allow us to optimize the lineshape broadening calculation. One of the conventional ways to do this is to approximate the Voigt profile by an analytical expression. Another commonly used approach is to reduce the number of lines by filtering “weak” lines out. The line wings can also be cut off to reduce the convolution kernel size. Some codes use multiple grids to have great accuracy near a line center and weak on the side. By combining all these methods, we can speed things up enough to compute a 30-M lines spectrum within minutes, where a naive algorithm would require days or weeks. A major disadvantage is that all of these techniques sever the quality of the spectrum. Moreover, even with these optimizations the code still relies on producing a lineshape for each individual line, and thus can never break $N_l * N_v$ complexity, where N_l is the number of lines and N_v the number of spectral points.

The RADIS approach

In addition to the standard optimization algorithms used in common spectroscopy codes, RADIS employs a unique numerical approach to analyzing the spectrum, named DLM (short for Distributed Linewidth Map). The DLM approach makes use of the fact that all lineshapes are more or less similar. In the case of identical lineshapes, the entire spectrum might be calculated by the convolution of a single lineshape and a stick spectrum, with only one datapoint per spectral line. The DLM method extends this approach to spectra with varying Gaussian and Lorentzian widths by making a discrete grid of Gaussian and Lorentzian widths, and distributing spectral lines over this grid, allowing the use of FFT-accelerated convolutions instead of summation over all individual lineshapes. The spectral grid over which the parameters are distributed, which we colloquially refer to as “the” DLM, can be calculated by simple vector calculations, which can therefore be easily parallelized. Doing so allows us to effectively linearize most of the equations. i.e., the 30 M convolution becomes a linear interpolation of a smaller number of convolutions that can be easily precomputed.

RADIS architecture

In order to understand how this project aims to improve RADIS' performance, we need to take a deeper look at the underlying structure of the broadening methods that are used in RADIS.

The architecture of RADIS can be found on the RADIS website and is reported in Figure 1 below:

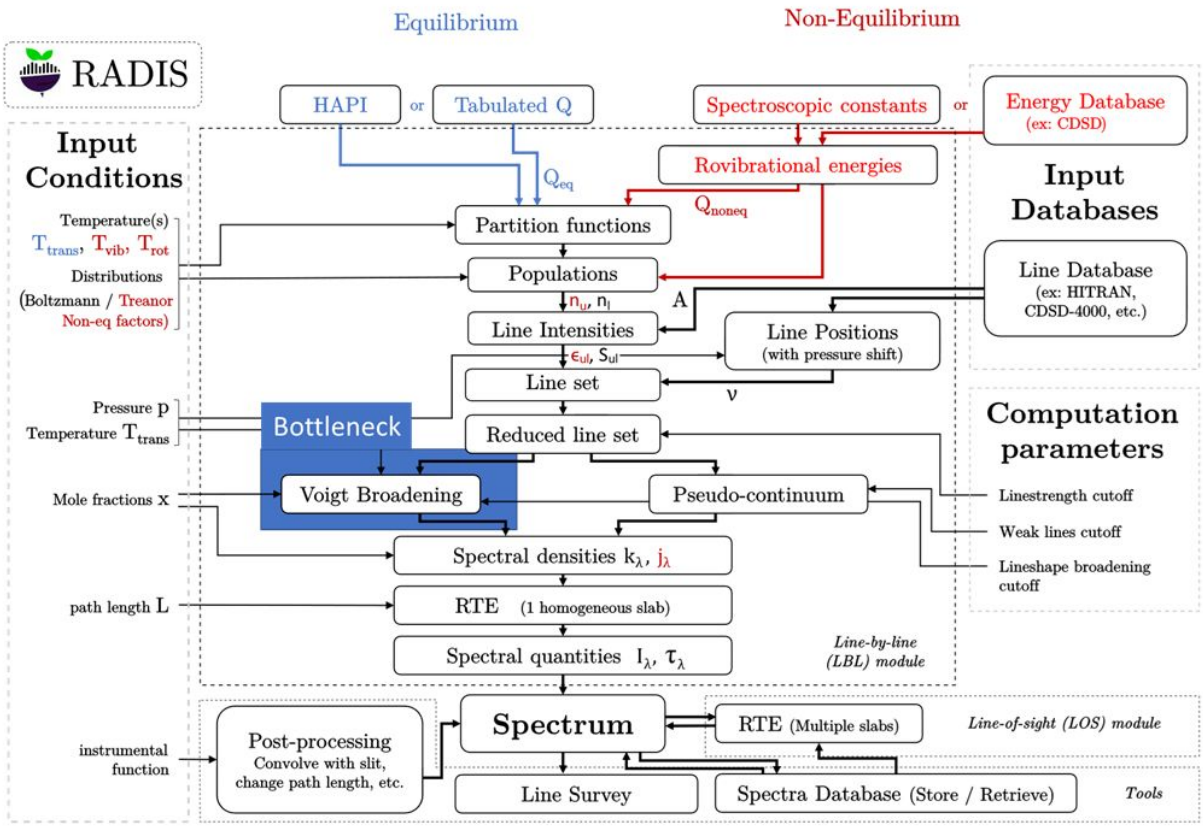


Figure 1 RADIS line-by-line calculation Architecture

The broadening step, the bottleneck for spectroscopy codes, is shown as *Voigt Broadening* in the above chart .

Most of the modifications in the Voigt broadening step will touch the broadening.py file of the radis lbl (line-by-line) module. It is important to note that Radis is a production code that can deal with

many input parameters. Some are physical parameters, and other are computation parameters. For instance:

- Equilibrium and nonequilibrium : whether the gas densities can be characterized with a single temperature (for instance: the air composition is well-known at any temperature).
- 1 CPU or multiple CPU calculations.
- Lineshape algorithm to use: standard, or the new DLM described earlier.
- Which convolution algorithm to use: FFT, integral convolution, or analytical approximation of the Voigt profile.

The broadening.py accounts for all these cases. As a first step to understand the project, the `broadening.py` file is described in Figure 2. Several functions, such as '_split_lineshape()', are all contained in `broadening.py`.

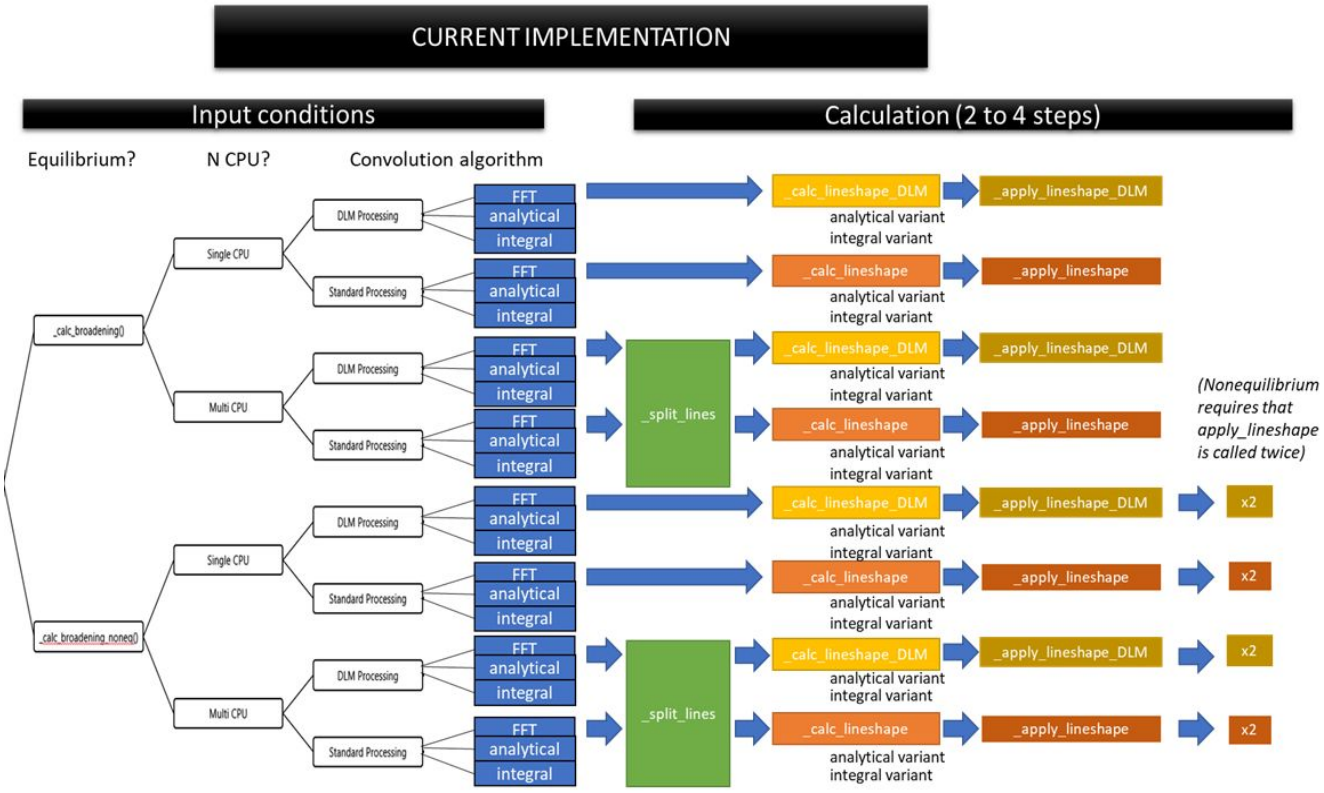


Figure 2 The different options in lineshape broadening calculation step

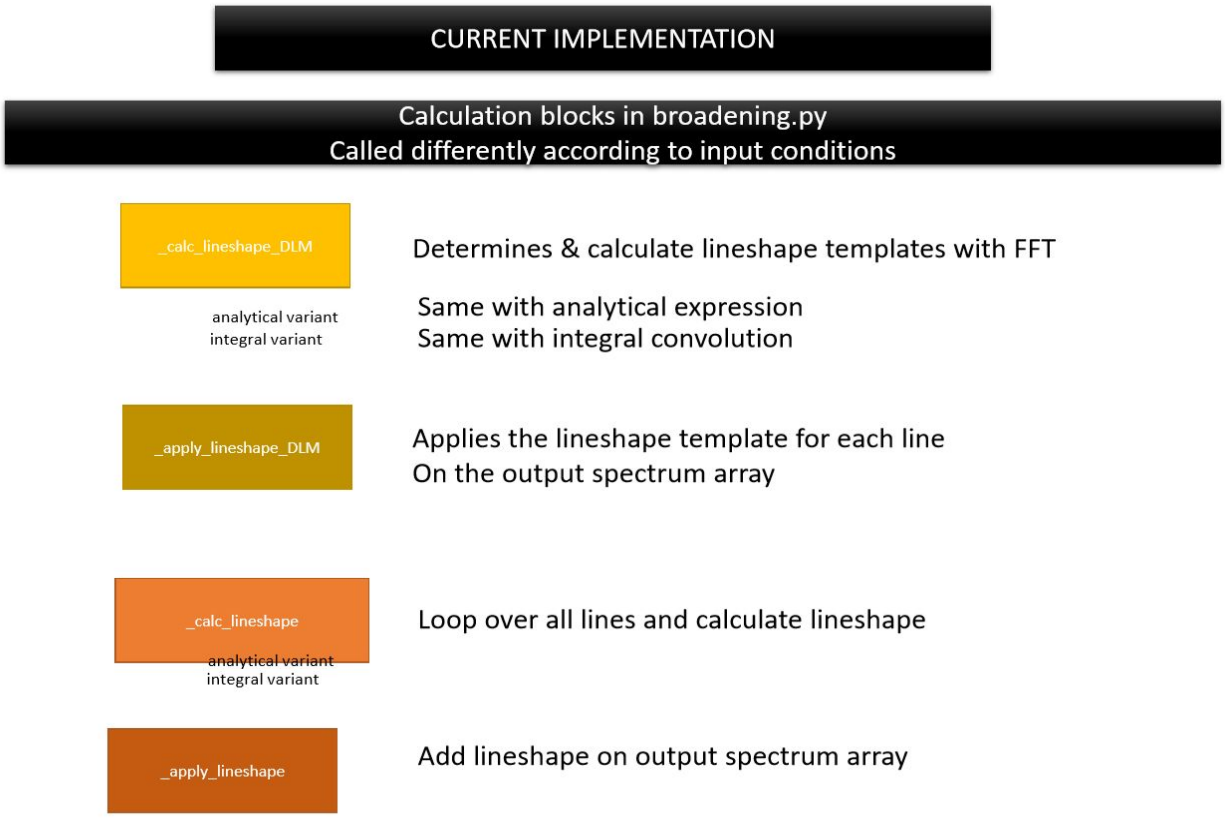


Figure 3 The different options in lineshape broadening explained

In the project, it will be important to be able to deal with all the input conditions mentioned in Figure 2. We will proceed in two steps. First we implement one set of input conditions, as in Figure 4. This will be done by first working on a sandbox (Weeks 1-2), then working on a fork of the production project (Week 3-4) and result in Evaluation 1.

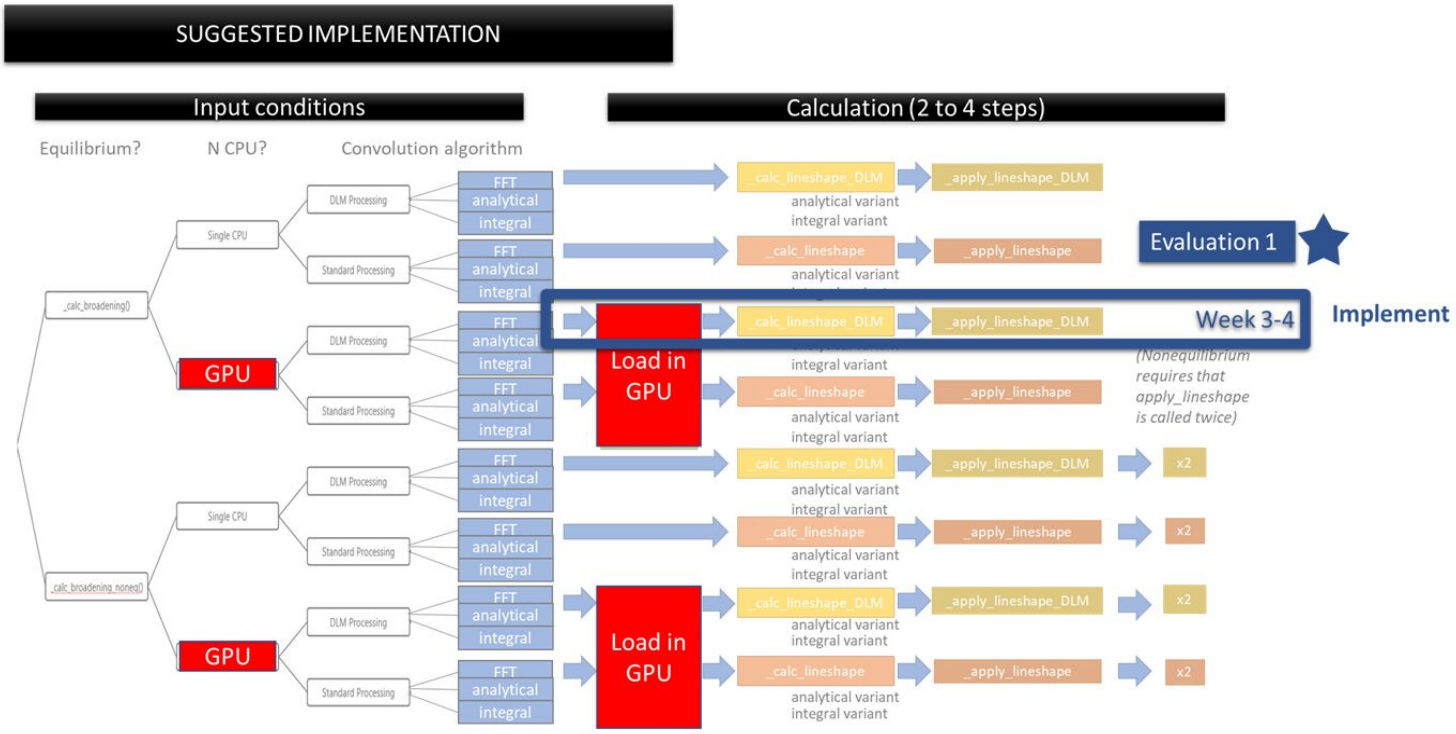


Figure 4 Suggestion of first set of input conditions to be implemented

Then we will work on the other input conditions, as in Figure 5. This will result in Evaluation 2.

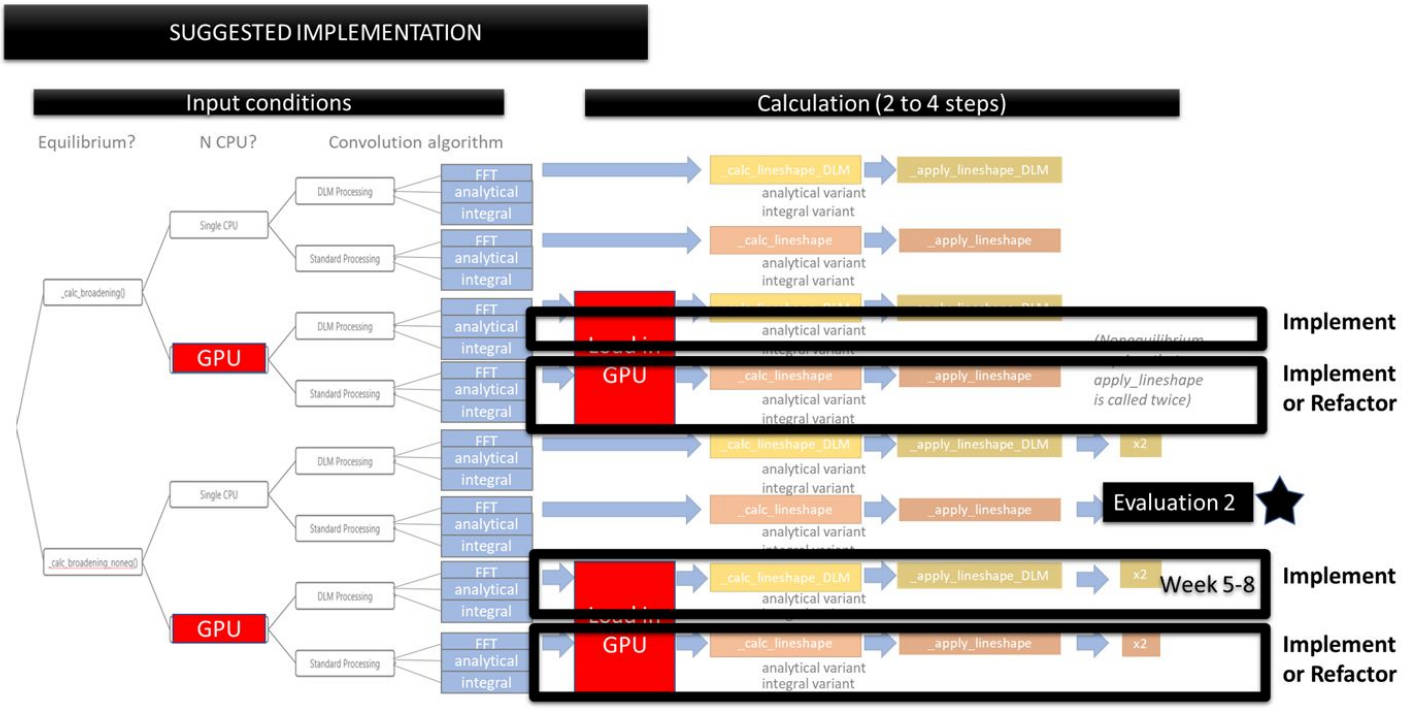


Figure 5 Suggestion of second set of input conditions to be implemented

The suggested weekly schedule will be described in the Timeline section. The next section describes some of the implementation details and the libraries used.

Parallelizing the Methods

A spectrum is built up of a large number of N individual spectral lines. For a given pressure, p, and temperature, T, four quantities of each line are calculated:

1. Position, ν_{dat}
2. Gaussian width, ω_G
3. Lorentzian width, ω_L
4. strength/intensity, S

Once we have these four parameters for each line, we can synthesize the entire spectrum.
The process of synthesizing the spectrum can make use of the GPU in two ways:

1. Use CPU to calculate the line parameters (v_{dat}, ω_G, ω_L, S) and use GPU to synthesize spectrum. However, for large databases (N>100M lines), the GPU bandwidth will be saturated, due to a slow memory transfer between host and device.

2. Load entirely the database file in the GPU, and calculate all four quantities needed, along with the spectra, on the GPU. This limits the database size to the GPU memory (~4GB on a standard GPU), and requires initialization time. However, this is compensated by fast calculation as each iteration is only limited by internal-memory-bandwidth. The only host-device communication at each iteration is limited to two transferts:

- Input: the pressure, p, and temperature, T.
- Output: a spectrum (size of the spectral axis, ~100k * 4 Byte).

We will focus primarily on the second solution, given its speed advantage. The program structure for the second option would look something like:

```
//initialize system
send_data_h2d(v0,da,S0,El,Eu,log_2gs,na,log_2vMm) //send database to device

//every iteration
For p,T in p_arr,T_arr:

      //Host:
      blocks = calc_blocks(p,T)             //calculate blocks
      send_constants_h2d(p,T,blocks)        //copy to device

      //Kernel 1:
      S,v0,wG,wL = calc_line_params(p,T)    //calculate parameters
      DLM = fill_DLM(S,v0,wG,wL)            //fill "DLM" matrix

      //Kernel 2:
      DLM_FT = fft_forward(DLM)             // forward FFT

      //Kernel 3:
      DLM_FT2 = DLM_FT * gW * gL            // multiply with lineshapes in fourier space

      //Kernel 4:
      I_out = fft_reverse(DLM_FT2)          //reverse FFT

      send_data_d2h(I_out)                  // send data back to host
```

Following this approach, we can compute the spectra quickly after a “slow” initialization. Looking at the code in detail, it’s clear that the majority of time would be spent in implementing kernel 1, whereas kernel 2 and 4 are standard functions which are already implemented in standard CUDA libraries. Kernel 3 on the other hand is fast enough to not make any significant difference towards the program’s performance.

The performance of kernel 1 is not limited by transfers between host and device, but by internal reads/writes to and from global memory. This leads to a problem as we need to add values to the DLM very often, which is a read/modify/write operation stalling execution for 100 clock cycles or more.

We will therefore make use of shared memory to first calculate smaller DLM's (on the block level). When this task is completed, the small DLM will be atomic-added to global memory. This leads us to the question: how to divide the lines over the different blocks?

Luckily, the database is sorted by line-position, and the memory of the DLM is also sorted by line-position, so we group lines approximately the same position in a single block, and add their parameters in a mini-DLM that lives in shared memory. When all lines in that block are processed, the mini-DLM will be added to the complete one that lives in global memory using a single atomic add.

Thus, we see that an important task in the implementation of this algorithm is coming up with new algorithms/heuristics to allocate the lines to the appropriate blocks. However, this further poses a problem, as the exact location of the line itself is a function of pressure. Thus, there is no way to ascertain whether all the lines will really fit in a given block. This in turn means that some lines will "spill", i.e., which need to be added to the big DLM in global memory directly.

The CPU-code, which calculates the blocks, returns a list of line indices where a new block should start. This list of indices is transferred to the GPU's constant memory, in addition to the pressure p and temperature T.

Finally, another major objective of the project would be to benchmark both the first and second parallelization options that are available against different input sizes, so as to understand which option works better for a specified input. The end goal would be to create a rule-of-thumb formula to decide which path to take depending on the input.

Timeline

Community bonding period

I would like to utilize this period by going through the original RADIS paper[1] in order to get a fundamental idea about the package and it's ideation. After that, I would like to go through the draft of the new algorithm[2] describing the DLM approach which will help me gain a better understanding of the new algorithms before I begin parallelizing them.

In addition to the everything mentioned above, I'd like to complete the following objectives during the community bonding period:

- Engage with the community and understand the motivation of spectroscopy users
- Training on emission & absorption spectroscopy
- Have set up a development environment, be familiar with open-source tools (GitHub / Git / Tests)
- Get used with RADIS architecture: review the interface change to calculate multiple molecules at the same time (#74)
- Learn about the details of the new technique used for line-of-sight spectra

Week 1, 2

The proof-of-work example that has already been implemented and benchmarked will be reproduced. This will also be time to further understand the code architecture. At the end of Week 2 the Implementation plan (Figure 5) is updated if necessary, and validated.

Week 3, 4

GPU acceleration for a specific broadening method will be implemented and benchmarked on personal fork and demonstrated to the mentors for potential improvements/integration with the main repository. This will also conclude my objectives set for the first evaluation.

Week 5

I plan on keeping the week immediately after Phase 1 evaluation as a **buffer period** to ensure that everything that has been implemented so far is in order and exactly the way it was intended. Additionally, if the mentors have some feedback regarding potential changes in the parallelized

code, this would be the time to work on it before we move on to other methods. Moreover, this would also be the time where I would have gained sufficient in-depth knowledge of the code to discuss the refactoring changes needed for the second phase evaluation.

Week 6

GPU acceleration is implemented for all methods in the main repository and tested on various different architectures.

Week 7, 8

The new GPU-compatible methods will be tested extensively. Additionally, thorough documentation of all the new methods and classes will be written to ensure the users can navigate through the new package.

This will complete the objectives that have been set for the 2nd phase evaluation of the project.

Week 9, 10

At this stage the package will consist of both the new, updated code along with other legacy portions that may or may not be necessary. Thus, after considering the situation at that time with my mentors, it will be decided which parts of the code have been rendered useless by the newest implementations and will subsequently be removed after proper discussion.

The proof of concept for line-of-sight spectra technique will be implemented.

Week 11,12

An iPython notebook will be created to document the performance benchmark on radis-benchmark. In addition to that, comprehensive tests and documentation will be generated for the RADIS and pushed to the main repository.

Week 13 + Stretch Goals

As mentioned in the 'Parallelizing the Methods' section, I would like to spend this week and any further time I could devote to RADIS in comparing and benchmarking the different ways to parallelize code, and see how their performance differs with different inputs. I would also like to use this time to implement the direct line-of-sight technique in the main repository, which would require some preliminary work in the LOS module, which I will do in week 11 & 12 if I'm done with the rest of the objectives.

ABOUT ME

Basic Information

Name	Pankaj Mishra
University	Indian Institute of Technology, Kharagpur
Major	Electronics and Electrical Communication Engineering
Minor	Computer Science and Engineering
Email address	pankajmishra1511@gmail.com
Github	pkj-m
LinkedIn	pkj-m
Timezone	IST (UTC +05:30)
Contact Number	+91 831-786-5280

Platform Details

OS	Ubuntu 18.04
Editor	PyCharm, Visual Studio Code

Background

I am a third-year undergraduate student in the department of Electronics and Electrical Communications Engineering at Indian Institute of Technology (IIT), Kharagpur. Mathematics and sciences have always been fascinating topics for me. I was introduced to the world of programming and specially drawn into it by the International Olympiad of Informatics (IOI) for which I prepared during my junior high school years.

I have been working with python for almost 6 years now, initially as a subject offered in high school to use it pretty much everyday now for various personal projects and scripts. Apart from Python, I am also well versed with parallel programming and CUDA. While my first introduction to using parallel programming and GPUs to accelerate numerical calculations was during my last internship with Julia Computing, I developed a solid, theoretical understanding of the subject in the following semester when I opted for a course on "High Performance Parallel Computing" at my university. I found the idea of using GPUs to perform calculations other than the traditional graphic processing they were initially meant for quite interesting (also peaked by Bitcoin farmers), and decided to pursue the topic further. Presently, I am an undergraduate research assistant working at the intersection of deep learning and high performance parallel computing using CUDA for my undergraduate thesis.

In addition to being familiar with these languages and frameworks, I also have past experience in developing and contributing to open source projects as a result of my previous internship with Julia Computing as a part of their "Julia Season of Code" program, where I was responsible for writing a library for automatic computation of sparse Jacobians for the Julia language from scratch. It was a wonderful experience for me and my work was well appreciated by the community. I am still involved with the community and regularly participate in their meetings to decide where the package is headed and participate in discussions over anything else they might be working on. The detailed summary of my work can be found here: https://nextjournal.com/pkj-m/a-summer-with-jacobians

Finally, I believe that I will be able to give GSoC my complete time and energy since the coding period greatly overlaps with our university's summer break. Since I have no other commitments, I can easily put in 30+ hours every week to complete the project, and more if I happen to be lagging behind schedule due to any unexpected issues.

Contributions to Radis/Sandbox

Since my project involves modifying an existing, actively used package, I decided to first start from a standalone, reduced version of a spectral line-by-line code to focus on building the GPU-prototype for the package and get it working as soon as possible. The repository below contains a minimal example of a line-by-line absorption code and will be used in the first phase of the implementation.

Link to the repository:

https://github.com/pkj-m/radis-lab

following which I will proceed to modify the production code.

Meanwhile I also got acquainted with the code base of RADIS by running the examples as well as tests for the package. Presently I am going through the code base in detail to understand the exact implementation of the algorithms I am supposed to modify.

In addition to this, I have also contributed to the sunpy repository with the following PRs:

1. https://github.com/sunpy/sunpy/pull/3776 [MERGED]
2. https://github.com/sunpy/sunpy/pull/3757 [MERGED]
3. https://github.com/sunpy/sunpy/pull/3835 [MERGED]

REFERENCES

[1] RADIS: A nonequilibrium line-by-line radiative code for CO2 and HITRAN-like database species, E. Pannier & C. O. Laux, doi.org/10.1016/j.jqsrt.2018.09.027

[2] A discrete integral transform for rapid spectral synthesis , D. v.d. Bekerom & E. Pannier (in preparation)

[3] NASA Code NEQAIR: E. E. Whiting, P. Chul, Y. Liu, O. Arnold, A. Paterson, C. Park, Y. Liu, J. O. Arnold, J. a. Paterson, NEQAIR96, Nonequilibrium and Equilibrium Radiative Transport and Spectra Program: User's Manual, Tech. Rep., 1996.

[4] JAXA Code SPRADIAN: Fujita, K., Takashi, A., and Abe, T., "SPRADIAN, Structured Package for Radiation Analysis:Theory and Application," 19