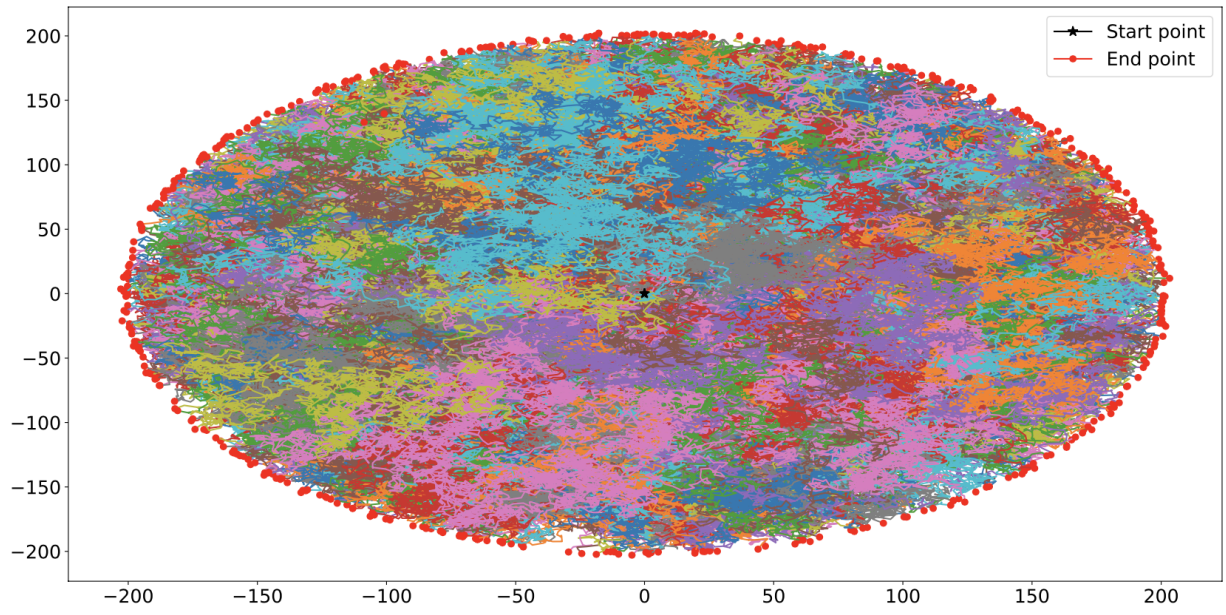
Parallelizing Monte Carlo Methods for the TARDIS SN Package

1. Abstract

The TARDIS SN package is an open-source Monte Carlo radiative-transfer code for modeling exploding stars. TARDIS SN uses Monte Carlo methods to track the random walk of photons after the supernova occurs. These photons are modeled computationally through probabilistic processes applying Monte Carlo methods. We first implemented a random walk for a single photon simply using python by creating a function that tracks the path of the photon with random scattering (change in its direction). The function was then updated to be able to track thousands of photons at a time and then plotted (see fig 1). The runtime for 1000 photons (we call packets) taking 10,000 random steps is 24.2 s. We then modified the functions to implement numba (a just-in-time compiler), which decreased the runtime to 433 ms under the same conditions. The next parallelization technique we applied was numba's cuda implementation to leverage the capabilities of a GPU. Our cuda implementation decreased the runtime to 45.7 ms. Both of the parallelization techniques we applied decreased the runtime, in the rest of this paper we will discuss the methods used that led to speeding up the functions.



1. Introduction/Background

It takes approximately 10 minutes to run a single TARDIS SN model. When astronomers want to fit their models, this requires iteratively re-evaluating the model with different parameters until they converge to a good solution. Fitting a model would require running the code many times and is a problem when that becomes hundreds of thousands of iterations. The TARDIS SN research team is tackling this problem by leveraging neural networks' ability to approximate functions with high precision using the TARDIS “emulator”. However, creating these training sets for the “emulator” requires thousands of models, so a faster version of TARDIS is required to create these training sets. Our goal is to write a simple random walk algorithm in python to use as a benchmark for TARDIS. We used two parallelization techniques, numba and cuda, to speed up the runtime of the simple random walk function. The runtimes of the serial, numba, and functions will be compared to the performance from a previous study, which saw a speed up of 1080x [1].

2. Method

2.1. Base code

The first step was to implement a random walk code in python. This was initially done by tracking the path of a single packet (photon) scattering n times. This code was then modified, so that the length of each scattering was variable. The final version of the serial code implemented a boundary that once the packet reaches the boundary it has escaped the supernova. The random walk function could be run for thousands of packets as would happen in a supernova, we implemented this simply using a for loop and then append function for lists.

2.2. Numba

The serial random walk for 1000 packets with each one taking 10,000 steps took 24.2 s to run. The python library Numba was the first parallelization technique we used to try and speed up our code. Numba is a just-in-time compiler that works best with numpy arrays, functions, and loops, which we used heavily in our serial code. This meant we did not need to rework major parts of our serial code and made it simple to speed up our code with Numba. Simply adding @njit infront of our functions sped up our code to 433 ms for 1000 packets.

2.3. Cuda

Same with the previous two versions, a single packet function was implemented to do the Monte Carlo simulation for one packet, and a main function was implemented to distribute packets into threads. We used two 2D numpy arrays to store coordinates for all packets, which is shared between all threads. Two arrays were initialized to negative infinite for future plotting. To make it compatible with numba cuda, we used

xoroshiro128p random generator to generate rng_stage first in the main function, and generated random number accordingly inside the single packet function. The single packet function will store the calculated coordinates array in the same index of calculated thread id. When the distance between the origin point and current position is larger than a specific limit, it will jump out of the loop and return to the main function.

2.4. Performance evaluation

We measured the performance in runtime and memory usage aspects. To evaluate the runtime, the *time* function was used before and after the main function and the difference was calculated as the output. To measure the memory usage, the *guppy* package [2] was used right after and before the timing (Figure 2.) and we only considered the total memory usage of the measured code.

```
begin = time.time() # begin timing
hp.setrelheap() # begin mesuring memory
main_function
print(hp.heap()) # end measuring memory
end = time.time() # end timing
```

Figure 2. Performance evaluation strategy

3. Results

We successfully implemented Monte Carlo simulation using numba and numba cuda, and the results showed great agreement.

3.1. Runtime

According to the runtime comparison table (Table 1.) and figure (Figure 3.), the numba package could help speed up by about 100 times, reducing from 130.8527 s to 1.5840 s for 5000 packets, by using LLVM based compiler to make high-level optimizations. This performance is similar with previous study, which found that Python with numba can reduce 99% of the computational time relative to the pure Python [3]. It's not surprising to see that base code and numba version both show a near-linear trend between the number of packets and runtime, as all packets are run successively.

Table 1. Runtime for base code, numba version, and cuda version

Number of packets	Base time (s)	Numba time (s)	Cuda time (s)
1000	24.8636	0.5608	0.0805
2000	51.6577	1.2235	0.0753
5000	130.8527	1.5840	0.2538

10000	261.81	3.1979	0.3289
20000	518.8404	5.9242	0.7453
50000	1306.4581	15.6866	2.0099

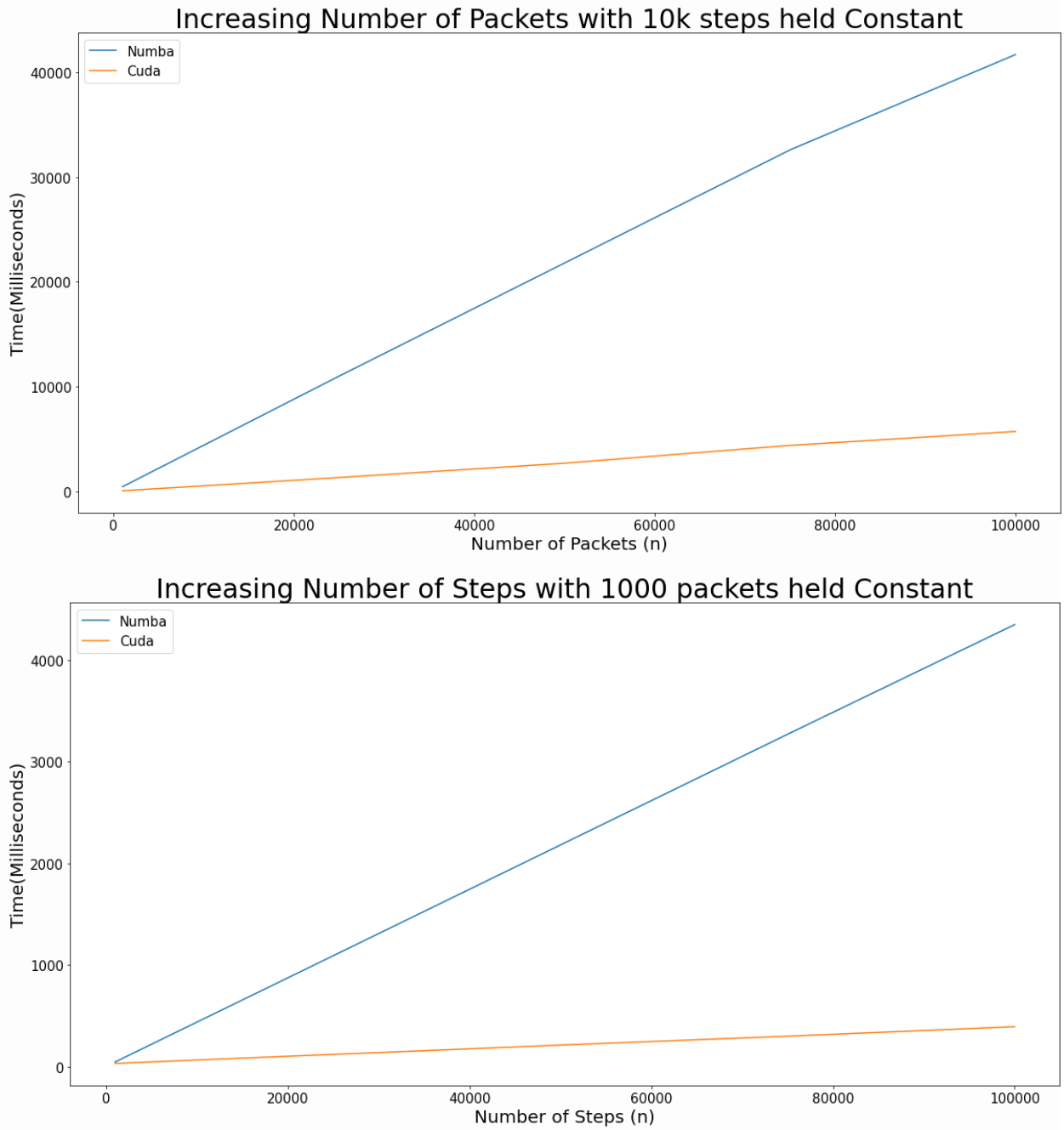


Figure 3. Runtime comparison between numba and cuda in variable steps and packets. The base code is too slow and doesn't show up in this figure.

By implementing the functions using cuda, the code was sped up by at least 7 times compared to the numba version and at least 300 times compared to the base version.

The difference was even larger when we were trying to apply to more packets. In the current setting (64 threads per block), we didn't achieve a competitive performance compared with previous study, which can achieve 1080 times faster [1]. And this performance couldn't be achieved when trying different threads per block numbers using the same code.

3.2. Memory usage

The memory usage comparison between the three versions of the code is shown in table 2. The numba package can help save memory dramatically compared to the cuda implementation and the base code. When computing large amounts of packets (e.g. 50000 packets), the cuda version took use of 200 times more memory to achieve 10 times faster compared with the numba version, which indicates that our code still needs to be optimized in terms of memory.

Table 2. Memory usage for base code, numba version, and cuda version

Number of packets	Base memory (MB)	Numba memory (MB)	Cuda memory (MB)
1000	12.546120	11.034031	96.642800
2000	28.288776	11.258031	176.642800
5000	66.724168	11.930031	416.642119
10000	136.010760	13.050031	816.539410
20000	266.426440	15.290031	1616.539414
50000	659.609160	22.010031	4016.538970

4. Discussion

Though we sped up the code successfully, there are still lots of things we can try. For the python with numba, it's easy to implement and we cannot do much more on that. The performance is good enough. In the aspect of cuda, due to the dependence between different stages for one packet, we cannot speed up the computation by making it parallel. We can try to minimize the transferring time between device and host. In this project, we are using a shared array to store all updated values, which might waste some time in waiting to write the data. We can try to assign an array to each thread and combine after computing and check if it can help reduce writing and transferring time. On the other hand, we are using cuda in a similar way with threading, so it may be also good for us to compare the runtime and memory usage between both.

5. References/Notes/Code

5.1. Code

Numba code

```
@njit()
def single_boundary_numba(n, o_x=0, o_y=0, radius=200, step_limit = 5):
    x=np.zeros(n)
    y=np.zeros(n)
    for i in range(1,n):
        theta=2 * np.pi * random.random()
        step=round(random.uniform(0, step_limit),2)
        x[i] = x[i-1]+ step*np.cos(theta)
        y[i] = y[i-1]+ step*np.sin(theta)
        distance = (x[i] - o_x)**2 + (y[i] - o_y)**2
        if distance > radius ** 2:
            return x[0:i], y[0:i]
    return x, y



@njit()
def main_numba(num_packets, steps):
    packets_x = []
    packets_y = []
    for i in prange(num_packets):
        x,y = single_boundary_numba(steps)
        packets_x.append(x)
        packets_y.append(y)
    return packets_x, packets_y
```

Cuda code

```
@cuda.jit
def single_boundary_cuda(n, num_packets, o_x, o_y, radius, step_limit, packets_x, packets_y, rng_states):
    idx = cuda.grid(1)
    if idx >= num_packets:
        return
    x=packets_x[idx]
    y=packets_y[idx]
    x[0] = 0.0
    y[0] = 0.0

    for i in range(1,n):
        theta = 2 * np.pi * xoroshiro128p_uniform_float32(rng_states, idx)
        step = xoroshiro128p_uniform_float32(rng_states, idx) * step_limit
        x[i] = x[i-1]+ step * math.cos(theta)
        y[i] = y[i-1]+ step * math.sin(theta)
        distance = (x[i] - o_x)**2 + (y[i] - o_y)**2

        if distance > radius ** 2:
            break



def main_cuda(num_packets, steps):
    packets_x = np.full((num_packets, steps), -np.Inf, dtype=np.float32)
    packets_y = np.full((num_packets, steps), -np.Inf, dtype=np.float32)
    threads_per_block = 64
    blocks = math.ceil(num_packets / threads_per_block)
    rng_states = create_xoroshiro128p_states(threads_per_block * blocks, seed=random.random())
    single_boundary_cuda[blocks,threads_per_block](steps, num_packets, 0, 0, 200, 5, packets_x, packets_y, rng_states)

    return packets_x, packets_y
```

5.2. Reference

[1] Erik Alerstam, Tomas Svensson, Stefan Andersson-Engels, "Parallel computing with graphics processing units for high-speed Monte Carlo simulation of photon migration," J. Biomed. Opt. 13(6) 060504 (1 November 2008) https://doi.org/10.1117/1.3041496

[2] pythonprofilers/memory_profiler. pythonprofilers, 2021. Accessed: Dec. 10, 2021. [Online]. Available: https://github.com/pythonprofilers/memory_profiler

[3] A. S. C. Rego and A. L. T. Brandão, “General Method for Speeding Up Kinetic Monte Carlo Simulations,” Ind. Eng. Chem. Res., vol. 59, no. 19, pp. 9034–9042, May 2020, doi: 10.1021/acs.iecr.0c01069.