# Algorithm Optimization and Parallelization in Astrophysics

**Leonardo Ferreira, Fabricio Ferrari**

Instituto de Matemática Estatística e Física,
Universidade Federal do Rio Grande, Rio Grande, RS, Brasil

## Introduction

In this work we compare several algorithm optimization and parallelisation techniques useful in the astrophysical context. To test the techniques, we apply it to measure statistical momenta of stellar orbits in model of galaxies. The momenta in question are density, mean velocity, velocity dispersion, velocity skewness, velocity kurtosis. These momenta are quantities projected in the line-of-sight, thus they can be compared with observables from galaxies, allowing us to infer the abundance of each orbit family in a given galaxy. In this way, we can build the phase space distribution function of the galaxy, a procedure called orbit superposition. In order to achieve a reliable result the algorithm need to be executed on several hundred orbits, so a small running time is essential. The core program was implemented in Python. The optimization is based in developing a C Module using the Python/C API, keeping the code nearly identical to its original and in the same time using techniques for parallelism available in C. This is in strong contrast with other techniques which require a great amount of code refactoring. We achieved a performance gain of 10 times the original running time which is proportional to the number of cores in the machine.

## Orbit Theory Review

The orbits we want to study can be integrate using the Hamiltonean formalism. The Hamiltonean per unit mass in cartesian coordinates is

$$H(\mathbf{p},\mathbf{r}) = \frac{1}{2}\left(p_1^2 + p_2^2 + p_3^2\right) + \Phi(x_1, x_2, x_3) \tag{1}$$

We can also write the hamiltonean in term of the homeoid vector $\mathbf{m}$,

$$\mathbf{m} = \varpi\,\hat{\mathbf{e}}_\varpi + \frac{z}{q}\,\hat{\mathbf{e}}_z, \qquad \dot{m}^2 \equiv \dot{\mathbf{m}}\cdot\dot{\mathbf{m}} = \dot{\varpi}^2 + \frac{L_z^2}{\varpi^2} + \frac{\dot{z}^2}{q^2}. \tag{2}$$

Here $\varpi, \varphi, z$ are cylindrical coordinates and $L_z = \varpi^2\dot{\varphi}$. In this way, the Hamiltonean is function only of $m$ and $\dot{m}$

$$H(m,\dot{m}) = \frac{1}{2}\dot{m}^2 + \Phi(m).$$

The total energy per unit mass

$$E = \frac{1}{2}\left(v_\varpi^2 + v_z^2\right) + \frac{L_z^2}{2\varpi^2} + \Phi(\varpi, z), \tag{3}$$

and the angular momentum $z$ component $L_z$ are **integrals of motion**, which restricts the region in the phase space available to the orbit. Equation (3) may be readed as a relation between three variables $(\varpi, z, v_z)$ and two integral of motion $(E, L_z)$. The quantity $\Phi_{\text{eff}} \equiv \Phi(\varpi, z) + L_z^2/2\varpi^2$ is called the effectived potential. The region available to the orbit is limited by the **zero velocity curve**.

$$z_{\text{zvc}} = \pm\, q\left\{\left[\frac{-G\,M}{E - \frac{L_z^2}{2\varpi^2}} - a\right]^2 - \varpi^2\right\}^{1/2}. \tag{4}$$

To map the galaxy phase space we must

- Choose an energy $E_i$
- To this energy corresponds an equatorial circular orbit with $\varpi_c$ and $v_c^2 \equiv \varpi_c(\partial\Phi/\partial\varpi)_{z=0}$ given by the relation $E = \frac{1}{2}v_c^2 + \Phi(\varpi_c, z = 0)$
- Calculate the correspondent maximum momentum $L_{z,\max,i} = \varpi_c\, v_c$
- Create a sequence of momenta $L_{z,ij} = \beta_j\, L_{z,\max,i}$ - For each $L_{z,ij}$, sample the radius $\varpi_k$, $k = 1, \ldots, N_\varpi$, on the correspondent $\text{ZVC}_{ij}$, between $\varpi_{\min}$ and $\varpi_{\max}$ (both solutions of Equation
- Calculate $z_k$ from Equation
- Leave the star at position $\varpi_k, z_k$, with velocities $v_\varpi = 0, v_z = 0$ and $v_\varphi = L_z/\varpi$ and integrate its orbit for several crossing times.

## Measurements made from orbit algorithms

The orbit measurement algorithm extracts from the observational data the following information:
Density $\Sigma(R)$, Velocity $v_{LOS}$, Velocity dispersion $\sigma_{LOS}$, Skewness $\xi_3$, Kurtosis $\xi_4$

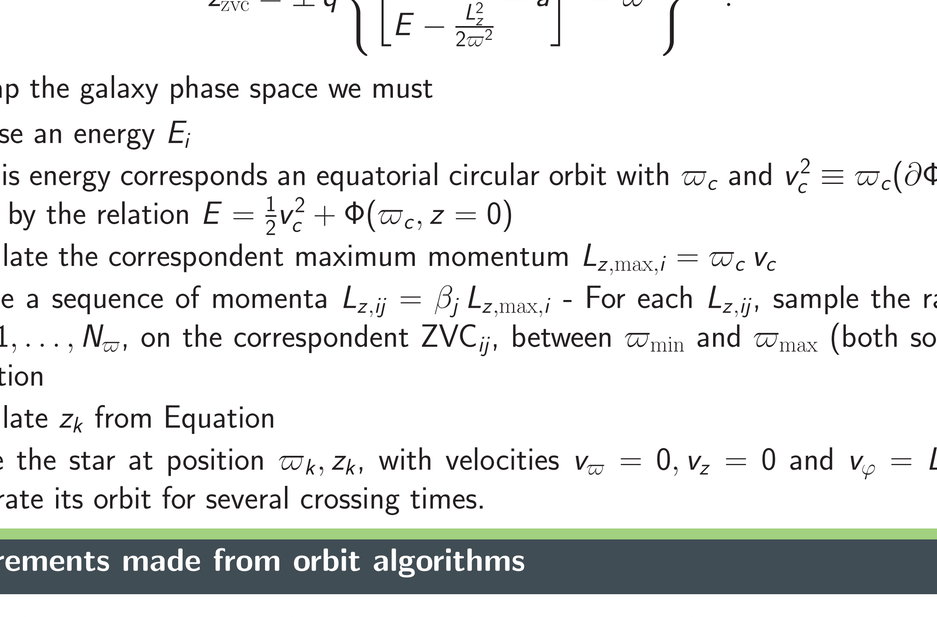


Figure: Orbit in 3D space (left) and its projected moments in the line-of-sight (right).

## Acknowledgements

Universidade Federal do Rio Grande (FURG)
Instituto Nacional de Ciência e Tecnologia de Astrofísica (INCT-A)

## CUDA

This API works as a medium to run algorithms in Nvidia graphics card. The code do not run on the GPU directly, **you need to crack down your algorithm** and send the kernel method to run on the device. The device is logically divided in a grid, blocks and threads. The grid has a number of blocks especified in the method call and each block has a number of threads also especified in the call. Each block has an shared memory, making all threads whitin it able to use it, but threads between blocks cannot communicate.

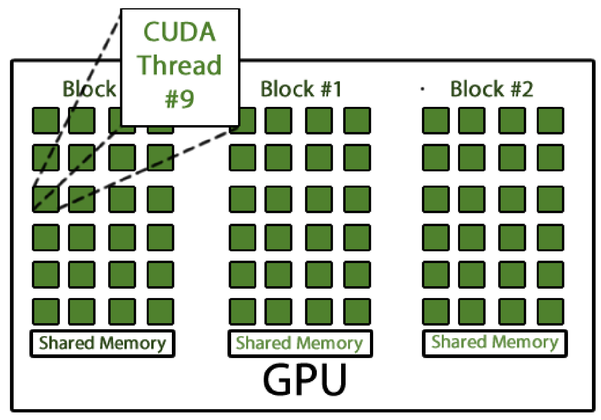


Figure: The virtual division of the graphics card with an <<<3,24>>>method call.

You need to transfer the data between the GPU and CPU, which slows down the performance. Also the debugging of CUDA programs can be so **much more time consuming** than Raw C or OpenMP's and it do not run in every graphical card, only in Nvdia CUDA enabled ones.

## OpenMP

The OpenMP API is **simple and fast** to use and develop. Its application is made easy with already existing code. All the parallelization is done by adding the API directives in the form of commentaries. You can setup and tweak each loop only with its keywords and the API will do the rest. In the code below **the only thing you need to add** is the `pragma omp parallel for` directive and compile the code with OpenMP libraries.

```
#pragma omp parallel for
for (i = 0; i < 100; i++)
{
    result += numbers[i];
}
```

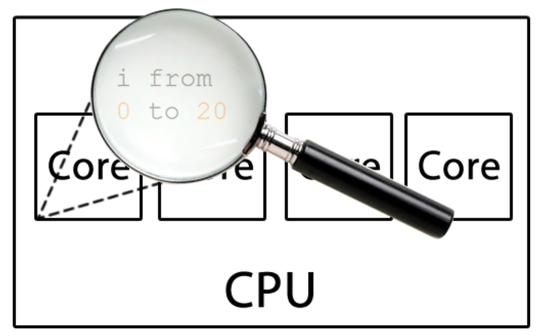


Figure: Example of OpenMP *pragma omp* directive usage and its sketch in the CPU's cores

## Benchmarking

We took some sample algorithms and implemented them in both technologies and in Raw C to have a control group. As the algorithms are simple, we executed them $10^4$ times with a for loop to have an mesurable time. The amount of data handled is also expressive, $a$ is an vector with $4 \times 10^5$ bytes of data, an $10^5$-dimension float vector.

Table: Benchmark Timings in $10^4$ iterations

| Algorithm | CUDA time | | OpenMP time | | Raw C time | |
|---|---|---|---|---|---|---|
| | **Development** | **Execution** | **Development** | **Execution** | **Development** | **Execution** |
| $\sqrt{a}$ | 2 hours | $\leq$ 0.5 s | 15 min | 1 s | 10 min | 2 s |
| $a^b$ | 2 Hours | $\leq$ 0.5 s | 15 min | 1 s | 10 min | 2 s |
| $/$ | 2 Hours | $\leq$ 0.5 s | 15 min | 1 s | 10 min | 2 s |
| $*$ | 2 Hours | $\leq$ 0.5 s | 15 min | 1 s | 10 min | 2 s |
| $\bar{a}$ | 4 Hours | $\leq$ 1 s | 30 min | 7 s | 20 min | 20 s |
| $\sigma$ | 4 Hours | $\leq$ 1 s | 30 min | 8 s | 20 min | 20 s |

## Conclusions

We showed that the CUDA's performance gain is better than OpenMP's, nevertheless it has an expensive development time due to its hardware-oriented API. The time expend with CUDA coding, refactoring and debugging is not worth the performance boost in the context of the algorithms we usually use in the astrophysics field, except for very particular cases. The OpenMP is so easy that its use do not add significative amount of time to the algorithm developing process. We can write C code almost unchangable to its original and be able to use all the power available in idle cores in the CPU. Also, we can wrap the OpenMP code to others programming languages easily. If you are willing for an fast and easy solution, OpenMP is the choice, but if you want to enhance an algorithm that you will use over and over again and its runtime is huge the CUDA API will serve you well.

## References

– T. J, Stuchi, *Sympletic Integrators Revisited*, Brazilian Journal of Physics, 32, p.958–979, 2002.
– M. Hénon, C. Heiles, *The Applicability of the Third Integral of Motion: Some Numerical Experiments*, 1964, A .J., 69, 73.
– R. Faber. CUDA, *Supercomputing for the Masses*
http://www.drdobbs.com/parallel/cuda-supercomputing-for-the-masses-part/207200659
– B. Barney. *OpenMP* [Internet]. Livermore (CA): Lawrence Livermore National Laboratory (LLNL);
https://computing.llnl.gov/tutorials/openMP/.