Flávio Manoel Santos Hemerli

Computação de Alto Desempenho Usando Numba para Cálculo de Agregação de Matéria

Alegre - ES, Brasil

2019

Flávio Manoel Santos Hemerli

Computação de Alto Desempenho Usando Numba para Cálculo de Agregação de Matéria

Trabalho de Conclusão de Curso apresentado a Universidade Federal do Espírito Santo, como parte dos requisitos para obtenção do título de Licenciado em Física.

Universidade Federal do Espírito Santo - UFES

Centro de Ciências Exatas, Naturais e da Saúde

Orientador: Prof. Dr. Roberto Colistete Júnior

Alegre - ES, Brasil

2019

Flávio Manoel Santos Hemerli

Computação de Alto Desempenho Usando Numba para Cálculo de Agregação de Matéria

Trabalho de Conclusão de Curso apresentado a Universidade Federal do Espírito Santo, como parte dos requisitos para obtenção do título de Licenciado em Física.

Aprovado em:

Comissão Examinadora:

Prof. Dr. Roberto Colistete Júnior
Orientador

Professor Dr. Mario Alberto Simonato Altoé

Professor Msc. Gabriel Lessa da Silva Lavagnoli

Alegre - ES, Brasil

Dedico este escrito a todos que tentaram me fazer acreditar que eu não conseguiria.

Agradecimentos

Agradeço a meus pais pelo apoio e por suportar que eu ficasse todo esse tempo na graduação.

Agradeço a meu orientador, Roberto Colistete Júnior, pela coragem de me orientar nessa pesquisa nada trivial para a graduação, pelas noites perdidas me ajudando com os códigos, e por treinar minha resistência a baixas temperaturas em sua sala.

Agradeço muito ao amigo Thiago Pires, que me abriu os olhos quando precisei, que me ajudou com quase tudo o que eu fiz, que me fez escrever esse texto e a quem devo a conclusão deste trabalho.

Por fim, não posso esquecer os amigos que me apoiaram como puderam, mesmo por vezes estando em situações piores que a minha. Gustavo, Artur, Eliza, Estevão, Lara, Hellen e Iggor, muito obrigado.

Resumo

A computação científica é muito utilizada na resolução de problemas de alta complexidade. Uma das limitações da prática é o tempo necessário para se aprender uma linguagem de programação e suas técnicas, tornando-a pouco convidativa em alguns casos. Nos últimos anos a linguagem Python tem crescido em aceitação entre pesquisadores devido a facilidade de aprendizado e ao crescente número de módulos científicos para ela. Não obstante, Python enfrenta certa resistência nas áreas onde se precisa de altíssimo desempenho, por ser há muito conhecida por entregar baixo desempenho comparado a C/C++. Utilizando do módulo Numba desenvolvemos um algoritmo n-body gravitacional para testar as ferramentas de paralelismo desse módulo, e com os resultados avaliamos se é possível obter alto desempenho em Python para aplicações científicas, encontrando desempenho centenas de vezes superior ao de Python puro usando paralelismo em CPU e milhares de vezes com paralelismo em GPU.

Palavras-chave: Python, Numba, Paralelismo, N-corpos.

Lista de tabelas

Sumário

Introdução

A computação científica é velha aliada de diversas áreas que requerem o cálculo de um grande volume de dados. Sistemas impossíveis de serem modelados em tempo hábil por mãos humanas (devido ao número de variáveis interagindo ou devido sua complexidade) exigem ferramentas muito poderosas para aproximação numérica de muitas ordens, que sejam capazes de calcular muito rapidamente. Nesse sentido, nem mesmo os computadores domésticos mais potentes são capazes de satisfazer essas necessidades.

No entanto, sendo possível usar uma grande quantidade de computadores interligados trabalhando num mesmo problema, podemos reduzir enormemente o tempo da modelagem e o custo computacional individual de cada máquina. A essa técnica chamamos paralelismo e é o princípio básico da supercomputação. Porém, montar um arranjo com vários computadores é caro e trabalhoso, limitando o acesso à tecnologia.

A pesquisa visa avaliar o desempenho e viabilidade do uso de Python com o módulo Numba no cálculo de agregação de matéria através de força gravitacional utilizando paralelismo de CPU e GPU. Utilizaremos a mecânica clássica com a ferramenta computacional Numba para modelar a atração gravitacional entre inúmeros corpos.

Os resultados produzidos apresentam o poder computacional da ferramenta e servir como base para a simulação de casos mais complexos e realistas.

No capítulo Motivações e Referencial Teórico explicamos o que é computação paralela, a tecnologia CUDA e a escolha de Python e Numba. A seguir, a Metodologia descreve como foi trabalhada a modelagem matemática do problema de n-corpos, a modelagem computacional das equações, o desenvolvimento do algoritmo para a simulação e os ambientes de execução para o algoritmo. Em Resultados e Discussões apresentamos os dados coletados pela simulação e os comparamos para obter medidas de desempenho. Em Considerações Finais tiramos conclusões sobre os resultados da pesquisa e ponderamos o que mais pode ser feito a partir dessa pesquisa.

1 Motivações e Referencial Teórico

1.1 A importância do aprendizado da linguagem Python

Segundo o parecer CNE/CES nº 1.304/2001, aprovado em 6 de novembro de 2001, que rege as diretrizes nacionais curriculares para os cursos de Física (PARECER..., 2001), utilizar os diversos recursos da informática, dispondo de noções de linguagem computacional, faz parte das habilidades gerais que devem ser desenvolvidas pelos formandos em Física independente da área de atuação escolhida. Dessa forma disciplinas básicas de programação são presenças constantes nas grades curriculares dos cursos de Física brasileiros.

Dentre as linguagens de programação trabalhadas em cursos de graduação, Python vem ganhando cada vez mais adeptos. Python (ROSSUM, 1995) é uma linguagem de alto nível originalmente desenvolvida por Guido van Rossum no final dos anos 80. Sendo considerada uma linguagem intuitiva e de fácil uso (FANGOHR, 2004), ela vem cada vez mais sendo indicada como uma primeira linguagem de programação a ser ensinada em disciplinas introdutórias.

Mais ainda, vários estudos (COSTA et al., 2017), (COLPO; FARIA; MACHADO, 2015), (MARQUES et al., 2011) tem obtido resultados positivos utilizando essa ferramenta com o público adolescente e, até mesmo infantil (BRIGGS, 2013).

Por outro lado, o poder e versatilidade da linguagem Python podem ser observados na indústria moderna. Grandes empresas como o Spotify (MEER, 2013), Netflix (RAPOPORT, 2013), Instagram (WEB..., 2016), Google (PYTHON..., 2012) entre outras, cada vez mais se utilizam dessa linguagem para variados fins.

Portanto, acreditamos que o estudo de Python não só tem um importante caráter introdutório no mundo da computação mas, simultaneamente, fornece aos estudantes uma poderosa ferramenta de trabalho.

1.2 Computação Paralela

Conforme as ciências avançam nos deparamos com modelos mais complexos para descrever a natureza. Esses modelos por vezes requerem soluções inviáveis de se obter com métodos simples como uma conta de punho ou cálculos computacionais simples, o custo humano e computacional seriam demasiados. Chamamos de computação paralela o processo da divisão de um problema em partes que podem ser resolvidas simultaneamente em paralelo. Consiste em modelar computacionalmente um problema de forma que diferentes partes dele sejam processadas individualmente e ao mesmo tempo, para então os dados de cada processamento serem manipulados para obter um resultado. Assim um grande problema que seria abordado por uma única entidade é dividido em n problemas menores abordados por n entidades. A computação paralela permite obter respostas mais rapidamente, e possui diversas aplicações em física dentro de cosmologia, astrofísica, física estatística, termodinâmica, mecânica de fluidos, estrutura da matéria e outras.

1.3 GPUs e a tecnologia CUDA

Uma GPU (*Graphics Processing Unit*, unidade de processamento gráfico) é o chip presente em placas de vídeo, muito utilizadas para jogos, que realiza todo o processamento das imagens que aparecem no monitor. Os cálculos para elementos gráficos de jogos são consideravelmente complexos e precisam ser feitos rapidamente para proporcionar uma experiência agradável aos jogadores, em termos de qualidade gráfica e responsividade. Por isso esses chips são equipados com uma grande quantidade de microprocessadores, vários núcleos, que, como no caso dos supercomputadores, dividem as tarefas entre si, transformando uma grande e complexa tarefa em inúmeras tarefas pequenas e simples(BELLEMAN; BÉDORF; ZWART, 2008). Graças à tecnologia CUDA da NVIDIA (NVIDIA..., 2019) podemos usar esses vários núcleos de uma GPU como se fossem os nós – os vários computadores interligados – de um supercomputador, reduzindo o custo e o espaço utilizado (BELLEMAN; BÉDORF; ZWART, 2008).

Com essa tecnologia podemos calcular e até exibir graficamente uma simulação gerada em tempo real de problemas que, de outra forma, seriam inviáveis de se

modelar. Um desses problemas é o problema da agregação de matéria através da força da gravidade. Uma vez que a gravitação newtoniana calcula a força atrativa entre apenas dois corpos (HALLIDAY; RESNICK; WALKER, 2000), um sistema ligeiramente mais complexo já exige um esforço desproporcionalmente maior para ser calculado (o problema de três corpos é famoso em gravitação). Quando pensamos em agregação de matéria no espaço estamos pensando em uma quantidade de objetos em ordens de grandeza muito maiores, um problema que, solucionado, nos ajuda a compreender a formação das estruturas do universo. Modelar manualmente esse problema seria, literalmente, o trabalho de uma vida. Felizmente, com a utilização de computação paralela, ele se torna um problema relativamente fácil e, graças à tecnologia CUDA, acessível.

Existem hoje várias soluções computacionais que utilizam a tecnologia CUDA, como a linguagem C/C++, para a qual a tecnologia foi desenvolvida, o software Wolfram Mathematica também permite paralelismo CUDA, PyCUDA e outros. Uma dessas soluções que está em desenvolvimento ativo no momento é o Numba, uma biblioteca da linguagem Python voltada para a utilização de CUDA em *scripts* Python de forma simplificada. Seu propósito é proporcionar a pesquisadores a possibilidade de utilizar paralelismo em CPU e GPU sem a necessidade de um aprofundamento técnico na linguagem C/C++, mantendo o código exclusivamente em Python, que é uma linguagem consideravelmente mais fácil de aprender e, por isso, muito popular entre pesquisadores.

1.4 Numba

A escolha pela ferramenta Numba se deve por ser relativamente a mais simples das que utilizam CUDA. Originalmente a tecnologia CUDA requer que o pesquisador utilize a linguagem de programação C/C++ e declare o *kernel* CUDA – uma estrutura que define como as operações serão paralelizados –, porém tal linguagem possui uma curva de aprendizado não amigável para pesquisadores que não a utilizam com frequência, se tornando um empecilho para aqueles que precisam de paralelismo mas não dispõem de tempo para aprender a linguagem e as especificidades da biblioteca CUDA. Já o Numba é desenvolvido para a linguagem Python, que vem

se popularizando muito entre pesquisadores na última década devido à facilidade de aprendizado. Numba tem seu desenvolvimento apoiado pela NVIDIA (NUMBA..., 2019), empresa criadora e responsável pela tecnologia CUDA e tem por objetivo facilitar a programação de paralelismo para o pesquisador que precisa focar em seus dados, sem perder tempo demais com a ferramenta. Numba permite que o desenvolvedor utilize paralelismo em GPU em Python puro sem a necessidade de declarar um *kernel*, e também permite a utilização de paralelismo em CPU com a adição de uma única linha de código, ambas características que agilizam a atividade de programação paralela.

Optamos por utilizar essas ferramentas no cálculo de agregação de matéria pois este é um problema recorrente no estudo de Astrofísica, uma área que muito utiliza recursos computacionais. O problema de n-corpos é recorrente em Física Computacional (GOZ et al., 2019) e é ponto de partida para inúmeros outros problemas em Astrofísica que ainda estão sendo investigados.

2 Objetivos

Essa pesquisa visou utilizar a linguagem de programação Python com o módulo Numba na resolução de um problema complexo de Física: o cálculo de agregação de matéria pela força gravitacional (também conhecido como problema gravitacional de n-corpos ou, simplesmente, *n-body*), utilizando diferentes métodos – com e sem paralelismo – a fim de comparar seus desempenhos e assim demonstrar a viabilidade da linguagem na computação científica de alto desempenho.

3 Metodologia

3.1 Descrição do problema

O problema escolhido foi o de simular um sistema isolado contendo um grande número de corpos pontuais dotados de massa inseridos num espaço tridimensional vazio, exceto por seus vizinhos, em distribuições homogêneas pré-definidas. Esses corpos deveriam interagir apenas através da força gravitacional exercida sobre cada um deles pelos seus pares. Damos a isso o nome problema de n-corpos gravitacional. Adicionalmente define-se que a velocidade inicial de cada corpo é nula, de forma que todo o sistema esteja estático no início da modelagem. Esse caso específico do problema de n-corpos gravitacional é conhecido como colapso frio, devido ao fato de que no início da modelagem toda a energia do sistema seja potencial e, devido á natureza atrativa da força gravitacional, o sistema tenda a colapsar em direção ao centro de massa.

Neste caso utilizou-se da mecânica newtoniana através da Lei da Gravitação Universal (HALLIDAY; RESNICK; WALKER, 2000), que calcula a força gravitacional $\vec{F_g}$ sobre um corpo devido a outro da seguinte maneira:

$$\vec{F_g} = G\frac{m_1 m_2}{r^2}\hat{r}.$$

A equação da Lei da Gravitação Universal calcula a força gravitacional exercida entre pares de corpos, onde G é a constante gravitacional, m_1 e m_2 as massas dos corpos envolvidos e r o módulo da distância entre eles, na direção de r. Para calcular a interação entre inúmeros corpos se atraindo mutuamente faz-se necessário somar as diferentes forças exercidas sobre um corpo devido a todos os outros, encontrando a força gravitacional resultante sobre tal corpo, assim podemos escrever a equação da força gravitacional como:

$$\vec{F_{g_j}} = \sum_{\substack{i=1 \\ i\neq j}}^{n} G\frac{m_j m_i}{r^2}\hat{r}.$$

Onde $\vec{F_{g_j}}$ é o vetor da força gravitacional resultante sobre um corpo j devido a todos os outros i corpos no sistema. Uma vez que um corpo não pode exercer força gravitacional sobre si mesmo, excluímos o caso em que $i = j$.

Através da equação da força gravitacional de cada corpo podemos obter as equações de movimento relativas para qualquer partícula j dentro do sistema, relacionando essa equação à segunda lei de newton:

$$\vec{F} = \frac{d\vec{p}}{dt} \Rightarrow \vec{F} = \frac{d(m\vec{v})}{dt} \Rightarrow \vec{F} = m\frac{d\vec{v}}{dt} \Rightarrow \vec{F} = m\vec{a}$$

Como a segunda lei de newton descreve a força resultante sobre um corpo como o produto entre sua massa e a aceleração sofrida por esse corpo, pode-se reescrever essa equação em termos da massa e da aceleração do corpo j:

$$\vec{F_{g_j}} = m_j\vec{a_j}$$

$$\Rightarrow \sum_{i=1, i\neq j}^{n} \frac{Gm_jm_i}{r^2}\hat{r} = m_j\vec{a_j}$$

$$\Rightarrow Gm_j \sum_{\substack{i=1 \\ i\neq j}}^{n} \frac{m_i}{r^2}\hat{r} = m_j\vec{a_j}$$

$$\Rightarrow G \sum_{\substack{i=1 \\ i\neq j}}^{n} \frac{m_i}{r^2}\hat{r} = \vec{a_j}$$

Uma vez que $\vec{a_j} = \frac{d\vec{v_j}}{dt}$ e $\vec{v_j} = \frac{d\vec{r_j}}{dt}$, podemos definir a posição e velocidade de qualquer um dos corpos em qualquer instante de tempo solucionando as equações diferenciais:

$$\vec{a_j}dt = d\vec{v_j}$$

$$\vec{v_j}dt = d\vec{r_j}$$

Mas deve-se atentar ao fato de que para cada intervalo infinitesimal de tempo dt é necessário resolver as as equações de movimento para todas as n partículas do sistema, para que seja possível encontrar esses valores para o próximo intervalo.

O caso específico do problema de três corpos já apresenta um nível de complexidade muito elevado. Para ordens mais elevadas que $n = 3$ não é possível encontrar uma

equação que descreva corretamente o sistema. O desafio não está em encontrar as equações de movimento, como demonstrado, mas sim descrever o sistema durante um longo período, visto que se deve recalcular todos os valores para os corpos a cada intervalo dt, sendo dt uma quantidade infinitesimal de tempo.

Pela mecânica clássica a gravidade é uma força de campo conservativa, o que significa que ela atua a distância e conserva a energia total do sistema. Como o problema abordado se trata de um sistema isolado com partículas interagindo apenas através de força gravitacional, infere-se que a energia total sempre será a mesma em qualquer instante de tempo, e como todas as partículas são providas de massa e não interagem de nenhuma outra forma, significa que as únicas energias presentes no sistema são as energias potencial gravitacional, devido à interação dos campos gravitacionais, e a energia cinética, devido ao movimento causado pela atração gravitacional, assim sendo podemos escrever a energia total como:

$$E = K + V,$$

onde E é a energia total do sistema, K o somatório das energias cinéticas de todas as partículas e V o somatório de todas as energias potenciais gravitacionais entre todos os possíveis pares de partículas.

$$E = \sum_{i=1}^{n} \frac{1}{2} m_i {v_i}^2 - \sum_{i<j=1}^{n} G \frac{m_i m_j}{||\vec{r}||}.$$

Essas equações permitem descrever o sistema de forma a acompanhar a evolução temporal da dinâmica das partículas em termos de energias, posição, velocidade e aceleração.

3.2 Levantamento

O desenvolvimento de algorítimos *n-body* é recorrente no estudo de cálculo numérico e física computacional (NYLAND; HARRIS; PRINS, 2007), portanto se decidiu que seria mais eficiente para a pesquisa iniciar com uma busca e estudo do que já foi feito e disponibilizado abertamente na internet. Usando uma ferramenta de busca foi possível encontrar inúmeros exemplos de algorítimos de simulações do

problema de n-corpos, especialmente no sítio GitHub, que é uma ferramenta de desenvolvimento colaborativo de software que permite a publicação de códigos e *scripts*.

Com intuito de definir um ponto de partida para o desenvolvimento do software, foram pesquisados algoritmos de simulações do problema de n-corpos. Essa pesquisa inicial serviu ao propósito de analisar o quanto já foi feito sobre o assunto, avaliar a dificuldade de se modelar o problema de n-corpos computacionalmente na linguagem de programação Python e encontrar modelos que servissem como base para o desenvolvimento de um software novo utilizando Python e Numba.

Dentre os algorítimos encontrados foram estudados apenas aqueles os quais era possível executar a simulação sem alterações no código, através do programa Jupyter Notebook presente no pacote Anaconda (que possui diversos softwares Python para computação científica, incluindo Jupyter Notebook, Numba, Numpy, etc).

O estudo desses algoritmos embasou os primeiros planejamentos para o desenvolvimento do software, pois pode-se notar o que havia na estrutura dos códigos que seria desejável implementar a fim de alcançar o objetivo da pesquisa – como a possibilidade de simular diversas distribuições de massa diferentes, definir por quanto tempo a simulação deveria ser executada, abordagens matemáticas de implementação simples –, e o que não era – como aproximação por método hierárquico, interação de outras forças além da gravitacional, orientação a objetos e animação em tempo real. A partir dessa análise definiu-se o escopo do desenvolvimento, limitando o propósito do algoritmo a simular um sistema de n-corpos interagindo gravitacionalmente a partir de um estado inicial estático.

3.3 Abordagem computacional

Para iniciar o desenvolvimento do algoritmo de modelagem do sistema, levou-se em consideração os exemplos encontrados na busca inicial. Atendo-se ao objetivo de criar uma simulação o menos complexa possível para fins demonstrativos, foi escolhida a abordagem de força bruta para o desenvolvimento do algorítimo Python. Essa abordagem consiste em traduzir literalmente os cálculos do modelo matemático mais simples para o sistema e repeti-lo para cada interação (NYLAND; HARRIS;

PRINS, 2007). Dessa forma faz-se com que o computador resolva a equação da força gravitacional para um dado corpo devido a outro tantas vezes quanto necessário, para cada corpo presente no sistema, e repita esse processo tantas vezes quanto se queira. Ou seja: para cada corpo j é calculada a força gravitacional devido a um corpo $i \neq j$, todas as forças sobre j são somadas para obter a resultante e então o processo é repetido para cada outro corpo no sistema.

A escolha dessa abordagem se dá por ser a forma mais simples de descrever as equações na linguagem Python. Uma vantagem decorrente da abordagem de força bruta é que a quantidade elevada de cálculos que o computador deve fazer acentua as diferenças de desempenho entre os métodos, tornando mais fácil de identificar quaisquer ganhos ou perdas de desempenho.

3.4 Escolha do método de aproximação numérica

Decorre do uso de método exaustivo para os cálculos do problema da impossibilidade de se expressar computacionalmente na linguagem Python uma quantidade infinitesimal de tempo, necessária para calcular velocidade e posição dos corpos e atualizar essas informações a cada interação, impossibilitando a conversão do modelo matemático exato em algorítimo Python. Para casos como esse usam-se métodos numéricos, que, embora não descrevam com total fidelidade a dinâmica do sistema, geram aproximações que permitem o estudo da evolução temporal de tal sistema, substituindo o valor infinitesimal, neste caso do intervalo de tempo entre iterações, por uma constante finita. Definindo-se qual é a margem de erro aceitável para o problema é possível escolher um valor para o intervalo temporal com um método de aproximação numérica que minimize o erro inerente de forma a permitir o estudo da evolução temporal do sistema. Assim possibilitando a modelagem computacional do problema de n-corpos de maneira que seja possível obter resultados coerentes.

A escolha do método mais adequado levou em consideração as vantagens e desvantagens de cada um para o objetivo da pesquisa. Problemas com o grau de complexidade muito alto tendem a gerar muito erro numérico que cresce exponencialmente conforme mais iterações são calculadas pelo computador. No caso do

problema de n-corpos a maior preocupação é com a conservação da energia total do sistema ao longo de muitas iterações. Visto que seria muito dispendioso verificar os erros específicos para cada objeto devido à aproximação numérica, utilizamos a conservação da energia total para calcular o erro da simulação (E/E_0). Dessa forma se faz necessário testar uma gama de métodos e relacionar quão bem cada um conserva a energia total do sistema ao longo de várias iterações com o custo computacional associado a ele (DANIEL; FOSTER-O'NEAL, 2012).

Existem muitos métodos disponíveis, e escolher o melhor entre eles exige uma certa quantidade de conhecimento sobre o comportamento da função e alguma quantidade de teste para adequação. Alguns métodos (como o método de Euler-Cromer, Verlet e Runge-Kutta de diferentes ordens) podem reduzir o erro enquanto aumentam o custo computacional, outros podem causar comportamento anômalo no sistema tornando a modelagem sem sentido (GIORDANO; NAKANISHI, 2006).

Contudo a definição do que seria o melhor método se resume à aproximação que mais adequadamente atende as necessidades do programador. Tendo em vista que toda aproximação gera erro, cabe ao desenvolvedor definir o que é uma margem de erro aceitável, quanto de trabalho é necessário para implementar um método que mantenha o erro dentro do limite aceitável e o custo computacional do método escolhido. Como a simplicidade foi definida como um fator importante para a pesquisa, optou-se por utilizar o Método de Euler. Esse método consiste em aproximar linearmente a função modelada:

$$v_{t+1} = v_t + a_t \Delta t$$

$$r_{t+1} = r_t + v_t \Delta t$$

Devido à aproximação linear o erro gerado cresce rapidamente com o tempo (GIORDANO; NAKANISHI, 2006). Pode-se minimizar esse efeito escolhendo-se valores muito pequenos para o intervalo de tempo entre interações, mas essa redução acarreta num aumento de igual valor no custo computacional da simulação, pois simular dez segundos do sistema a intervalos de um segundo exige dez interações, e o mesmo tempo de simulação a intervalos de $0,1$ segundos exige dez vezes mais interações, isto é, dez vezes mais contas.

Esse método foi escolhido porque é possível reduzir o erro facilmente ao reduzir a constante temporal e o efeito colateral do aumento de custo computacional pode ser ignorado uma vez que um dos objetivos da pesquisa é a comparação de desempenho, permitindo que tal aumento seja usado a favor do estudo ao aumentar a diferença de tempo de execução entre métodos mais eficientes e menos eficientes, e também sendo possível a redução do número de iterações do algorítimo para reduzir o custo computacional elevado.

3.5 Desenvolvimento do algoritmo

Para o desenvolvimento do algoritmo da simulação utilizou-se a plataforma científica Anaconda, que contém diversos pacotes para programação científica em Python, optando-se pela versão 3 desta linguagem uma vez que se tornou padrão para desenvolvimento Python, oferecendo maior compatibilidade de software e hardware e melhorias de desempenho, em detrimento da versão 2, que os desenvolvedores estão deixando de suportar e por isso oferece menos pacotes e software mais antigos. Desta plataforma foram utilizados os softwares e módulos Jupyter Notebook, Numpy, Numba e as bibliotecas **math** e **timeit** contidas no Python 3.

O Jupyter Notebook é uma ferramenta de programação que oferece diversos recursos para o desenvolvimento de códigos de programação interativos em Python e R, permitindo escrever e executar células de código uma a uma, inserção de texto, formatação em linguagem Markdown e saídas interativas. O uso dessa ferramenta permite o desenvolvimento do algoritmo em partes visualizando os resultados conforme são necessários, facilitando a correção de erros e o andamento da codificação (PERKEL, 2018).

O Numpy é um módulo Python utilizado para cálculo numérico, ele permite acelerar cálculos em Python através das várias ferramentas que possui, como vetores, matrizes, funções e tipos de variáveis próprias, criadas com esse propósito. Esse módulo é necessário para o uso de operações com vetores dentro das funções necessárias ao código, uma vez que permite criar variáveis do tipo "np.array"não disponível em Python puro. Diferente das listas do Python que, como toda estrutura em Python são tratadas como objetos, os *numpy arrays* se assemelham mais aos

vetores de linguagens de programação compiladas, e, visto que Numba compila parte do código Python para execução, algumas estruturas utilizadas funcionam melhor com o uso de Numpy, como atestado na própria documentação de Numba.

Numba é a peça de software mais importante desta pesquisa. Numba traz para a linguagem Python um compilador JIT (*Just In Time*) que permite acelerar a execução do código, paralelismo em CPU e paralelismo em GPU usando a tecnologia CUDA (NUMBA. . . , 2019). Com Numba podemos testar e comparar o desempenho do software executado de diferentes maneiras, Python puro, Python com Numpy, Python com Numba sem paralelismo, com paralelismo em CPU (com e sem Numpy) e paralelismo em GPU com CUDA. Utilizando todos esses métodos se faz possível determinar com qual deles se obtêm o menor tempo de execução para a simulação do problema de n-corpos e se o objetivo de criar um software simples que resolva de forma eficiente um problema complexo utilizando paralelismo pode ser atingido.

O desenvolvimento do algoritmo começou pela descrição matemática do problema. Optou-se por explicitar em três coordenadas as grandezas vetoriais do problema, sendo elas força, posição, velocidade e aceleração. Embora a prática aumente o volume de código escrito ela torna o *script* mais legível e facilita a manipulação dos valores das variáveis. A modelagem do problema requer que algumas constantes sejam estipuladas para os cálculos, sendo elas a constante gravitacional, o intervalo de tempo das interações e uma constante de atenuamento para a força gravitacional. A Constante Gravitacional é uma quantidade experimental presente na Lei da Gravitação Universal da mecânica newtoniana, ela possui valor de $6,67408 \times 10^{11} m^3/Kgs^2$ (HALLIDAY; RESNICK; WALKER, 2000), porém, no estágio inicial de desenvolvimento da simulação foi utilizado o valor 1 para tal constante por simplicidade. A constante de atenuamento é um valor que se insere no cálculo da força gravitacional a fim de evitar divisões por zero no evento de duas partículas do sistema ocuparem a mesma posição no espaço (NYLAND; HARRIS; PRINS, 2007). A inserção de estruturas condicionais ao código (*if* e *else*) permite evitar mais uma variável geradora de erro numérico, no entanto o uso dessa constante é superior e necessário porque estruturas condicionais alteram o fluxo de execução do código, fazendo com que diferentes casos sejam executados em tempos diferentes. Ao utilizar paralelismo, essas diferenças causam assincronia na execução, o que pode

acarretar diversos problemas. Em alguns casos, tentar compilar uma função com essas condicionais através de Numba não é possível. Uma vez que o modelo assume corpos pontuais, providos de massa porém sem volume (ou outra característica que impeça que mais de uma partícula ocupe a mesma posição), é necessária a inserção de tal constante para evitar divisões por zero e manter o tempo de execução.

$$\vec{F}_g = G\frac{m_1 m_2}{(||\vec{r}||^2 + \epsilon^2)}\hat{r},$$

onde ϵ é a constante de atenuamento.

A primeira etapa de desenvolvimento do modelo computacional para o cálculo da atração gravitacional culminou no seguinte código:

```
def gravitationalNbody(r, v, a, m, nobj, ninter, dt):
    for inter in range(ninter):
        for i in range(nobj):
            a[i][0] = 0.0; a[i][1] = 0.0; a[i][2] = 0.0
            for j in range(nobj):
                if i != j:
                    # calculate a_ij acceleration
                    dx = r[j][0] - r[i][0]
                    dy = r[j][1] - r[i][1]
                    dz = r[j][2] - r[i][2]
                    dsq = dx*dx + dy*dy + dz*dz + sft*sft
                    gA = (G*m[j])/(dsq*math.sqrt(dsq))
                    # sum a_ij_x to a_i_x, etc
                    a[i][0] += gA*dx
                    a[i][1] += gA*dy
                    a[i][2] += gA*dz
            # update the v_i velocity
            v[i][0] += a[i][0]*dt
            v[i][1] += a[i][1]*dt
            v[i][2] += a[i][2]*dt
        for i in range(nobj):
            # update the r_i position
            r[i][0] += v[i][0]*dt
            r[i][1] += v[i][1]*dt
            r[i][2] += v[i][2]*dt
```

A constante de atenuamento pode ser tão pequena quanto se queira, porém, se ela for muito pequena, as limitações computacionais acerca de precisão (quantidade de algarismos significativos), e tipo de variável podem acarretar em arredondamentos e o reaparecimento de divisões por zero, se ela for muito grande o erro gerado por sua inserção aumenta proporcionalmente. Determinou-se então o valor arbitrário $\epsilon = 10^{-10}$ na etapa inicial de desenvolvimento.

Para a constante do passo de tempo, que determina quanto tempo deve se passar entre duas interações do sistema os critérios de escolha são similares. Como decidiu-se pelo uso do método de Euler para a aproximação numérica, quanto menor o valor dessa constante mais realista se torna o modelo porém tanto mais se necessita de recursos computacionais, quanto maior o valor maior o erro gerado e tanto menos confiáveis se tornam os dados. Uma constante temporal muito pequena implica em resultados mais realistas e em um custo maior para observação da evolução do sistema, enquanto uma muito grande implicaria em uma maior dificuldade para averiguar se a simulação funciona corretamente. Com isso em mente optou-se por manter o valor do passo de tempo em 10^{-5} no início do desenvolvimento.

Para garantir que o modelo calcularia corretamente todos os passos da simulação, decidiu-se implementar um calculo para a energia total do sistema, uma vez que o mesmo é isolado e os corpos interagem apenas por uma força de campo conservativa. Dessa forma, pode-se calcular a energia total do sistema no instante inicial e comparar com a energia total final, fisicamente, para um problema como esse, elas devem ser iguais, porém existe a geração de erro devido as aproximações usadas no método numérico e devido à constante de atenuamento. Portanto, ao invés de buscar a igualdade entre as energias final e inicial, esperou-se que a diferença entre elas, isto é, o erro, estivesse dentro de um limite tolerável, estabelecido arbitrariamente como sendo 5%. Como o modelo se trata de uma simulação n-corpos de colapso frio, todas as partículas do sistema iniciam a simulação com velocidade nula, incorrendo em uma energia total igual a soma de todas as energias potenciais. O seguinte código foi adicionado à função anterior com essa finalidade:

```
...
    K = 0.0; U = 0.0
        for i in range(nobj):
```

```
            # Sum the total kinectic energy
            K += 0.5*m[i]*(v[i][0]**2 + v[i][1]**2 + v[i][2]**2)
            for j in range(nobj):
                if (i < j):
                    # Sum the total potential energy
                    U += (-G*m[i]*m[j])/math.sqrt((r[j][0] - r[i][0])**2 +
                                                  (r[j][1] - r[i][1])**2 +
                                                  (r[j][2] - r[i][2])**2 +
                                                  sft**2)
    return K, U
```

Para simplificar o modelo, adotou-se também o valor 1 para todas as massas de todos os corpos.

Para dispor as massas no espaço da simulação, inicialmente foi utilizado um método pra criá-las aleatoriamente. No entanto um sistema com distribuição aleatória não pode ser reproduzido. Para poder comparar os dados entre diferentes métodos e ambientes de software e hardware se faz necessário que se execute a simulação de uma mesma situação em todos os casos. Por isso adotou-se uma distribuição cúbica de massas para o sistema, por ser fácil implementar, tanto no quesito da parametrização quanto na configuração da quantidade de corpos.

Com a primeira versão pronta executando Python puro os primeiros testes foram feitos. Esses testes tiveram por finalidade averiguar se o algoritmo era capaz de calcular corretamente vários instantes do sistema, tendo o cálculo do erro da energia total como parâmetro para a qualidade das aproximações. Os valores das constantes temporal e de atenuamento foram ajustados diversas vezes, sendo reduzidos ou aumentados várias ordens de grandeza por vez. Chegou-se à conclusão de que o valor de $0,01$ para a constante temporal seria mais razoável. Esse valor mantém o erro dentro do aceitável por uma quantidade razoável de interações, ao mesmo tempo que permite uma evolução temporal notável do sistema dentro desse mesmo período, portanto o valor foi mantido até o fim do desenvolvimento.

Após desenvolver o algoritmo em Python puro capaz de simular o sistema proposto, seguiu-se a implementação dos métodos de otimização. Primeiramente com a implementação de Numpy, convertendo as listas de variáveis que representam os vetores de posição, velocidade e aceleração em *numpy arrays*, um tipo de variável

mais apropriado para operações com vetores.

Utilizando Numba foram criadas versões do código com e sem o uso de Numpy para efeito de comparação. Numba permite compilar funções, que podem ser executadas com ou sem paralelismo, em CPU ou GPU, uma versão do *loop* principal do algoritmo foi criada para cada possível combinação resultante de todas essas possibilidades, atentando às particularidades de algumas delas. Por fim o documento ".ipnb"do Jupyter Notebook continha versões em Python puro, Python Numpy, Python com Numba CPU, Python Numpy com Numba CPU, Python com Numba CPU e paralelismo, Python Numpy com Numba CPU e paralelismo e Python Numpy com Numba GPU via CUDA. O caso do uso de CUDA através do módulo Numba só é possível com o uso de *numpy arrays*, essa é uma particularidade devida as diferentes formas que Python e CUDA trabalham com grupos de variáveis (genericamente chamados de vetores). As funções destinadas a execução em CPU não apresentam muitas diferenças entre si, bastando usar o decorador "jit"de Numba e suas diferentes configurações, e ao uso de variáveis de tipos diferentes. Em resumo, paralelizar o código em CPU através de Numba requer o acréscimo de uma única linha de código antes da função a ser paralelizada.

```
@njit(parallel=True)
def gravitationalNbody(r, v, a, m, nobj, ninter, dt):
    for inter in range(ninter):
...
```

O algoritmo Numba com CUDA é um caso a parte no desenvolvimento. CUDA trabalha com métodos próprios diferentes dos de Python para computar funções. Utilizando Numba é possível automatizar muitas das configurações típicas de um código CUDA/C++. Primeiramente, ao declarar a função para o decorador "cuda.jit"é preciso especificar o tipo de variável de cada parâmetro da função no decorador. Por limitações da ferramenta foi necessário usar precisão dupla para os valores de ponto flutuante, o que incorre em um menor desempenho e maior precisão. Em seguida foi necessário declarar a geometria do *kernel*, para que o programa utilize corretamente os núcleos da GPU, que consiste em descrever a quantidade de *threads* em um bloco e a quantidade de blocos por grade (NUMBA...,

2019) (NVIDIA..., 2019). A maior diferença entre CUDA e os outros métodos se mostrou na declaração dos laços que iteram para calcular a aceleração sobre cada corpo. Essa ferramenta é capaz de iterar um laço por *thread* disponível paralelamente, então para obter a vantagem do uso dessa ferramenta os laços devem ser declarados paralelamente, usando a sintaxe específica de Numba para isso, ao invés de sequencialmente como nos outros casos.

```
@cuda.jit("(float64[:,:], float64[:,:], float64[:,:], float64[:], int32, float64)")
def gravitationalNbody_a_v_ij_NumbaGPU64(r, v, a, m, nobj, dt):
    i = cuda.grid(1)
    if i < nobj:
        a[i][0] = 0.0; a[i][1] = 0.0; a[i][2] = 0.0
        for j in range(nobj):
            if i != j:
...
```

A forma como o paralelismo CUDA funciona através de Numba exigiu que a função única que foi usada nos cálculos em CPU fosse separada em três funções diferentes, o laço principal do cálculo de forças, a atualização da posição das partículas e o cálculo da energia foram separados em três funções, cada uma compilada para a GPU separadamente, e uma quarta função, compilada para paralelização em CPU pelo Numba, foi criada para chamar as três primeiras de dentro dela.

Com tantas funções diferentes e tantos parâmetros possíveis de serem alterados, se mostrou necessária a criação de uma função dedicada a invocar e atribuir valores aos parâmetros das outras funções. Tal função recebeu o propósitos de criar a distribuição de corpos no sistema, atribuir suas massas, definir os valores dos parâmetros ao chamar uma função da simulação, fazer a chamada de função, calcular e exibir o erro da energia total do sistema, calcular o tempo de execução de cada função chamada.

Ao chamar essa função pode-se definir qual função da simulação deve ser chamada por ela, o tamanho da aresta da distribuição cúbica (que por fim define a quantidade de objetos), o número de passos de tempo a serem calculados, o valor do passo de tempo, se a função deve ou não usar Numpy, se ela é ou não compilada, se é necessário calcular a energia total e se é necessário exibir as listas de velocidade,

posição e aceleração para todos os objetos do sistema. Esses parâmetros permitem controlar alguns aspectos importantes da simulação. O tamanho n da aresta do cubo define a quantidade de objetos presentes na simulação tal que o número de objetos seja igual a n^3, e cada objeto adicionado à simulação aumenta o custo computacional dela. O número de interações ou passos de tempo define quanto tempo do sistema deve ser simulado e também interfere no erro e custo computacional, assim como o passo de tempo. Os quatro últimos parâmetros existem para atender às particularidades de cada função, permitindo trocar entre usar e não usar os "numpy arrays", que afetam o desempenho, explicitar a compilação de funções do decorador "jit"e realizar *debug*. O marcador de compilação é especialmente relevante na chamada de funções que utilizam dos recursos do Numba porque essas funções serão compiladas antes de executadas e esse processo requer tempo e recursos computacionais. Utilizando o marcador de compilação é possível chamar a função duas vezes, sendo ela compilada e executada na primeira e apenas executada na segunda, eliminando o tempo de compilação na segunda execução. O cálculo da energia total também pode ser ligado ou desligado, uma vez que seu propósito é analisar o erro em cada execução, ao verificar que a simulação não possui problemas ele pode ser desligado para economizar recursos computacionais, o mesmo vale para a exibição a cada *loop* dos vetores aceleração, velocidade e posição de cada objeto.

3.5.1 Truques de otimização

Ao longo do processo de desenvolvimento se percebeu que alguns cuidados deveriam ser tomados ao se utilizar algumas estruturas de programação com Numba. Em certos casos, testes com variações dos métodos levaram à descoberta de "macetes"que permitem uma melhor escrita do programa, em termos de resultados e da correta utilização dos recursos do *software*, eles estão citados aqui:

O paralelismo de Numba não funciona bem com estruturas condicionais (*if* e *else*) e deve-se evitar seu uso dentro das funções que serão compiladas.

A fim de melhorar o desempenho do programa, deve-se evitar o uso excessivo de variáveis, dando preferência ao uso de valores explícitos sempre que possível. Mesmo que armazenar vários em variáveis torne o código mais legível e fácil de entender, essa prática leva a redução de desempenho do código.

Deve-se também evitar o uso de variáveis globais e em seu lugar fazer uso de listas como argumentos de função. Dessa forma é possível modificar os valores desejados fazendo uso correto das funcionalidades de Numba (que é otimizado para uso com listas) e sem causar problemas de execução.

As listas usadas para paralelismo em CPU podem ser geradas em Python puro ou com Numpy e podem possuir quantas dimensões forem necessárias ao problema. Para paralelismo em GPU Numba só aceita listas geradas com Numpy (*numpy arrays*).

Ao utilizar paralelismo em GPU, é necessário usar tuplas para atribuir valores dentro da função.

3.6 Adequação e testes

Após o término dos mecanismos de simulação, as constantes que possuíam valor 1 adimensional para testes tiveram esses valores substituídos por outros com sentido físico. A constante gravitacional teve seu valor real utilizado e aos corpos se atribuiu a massa de $10^9 Kg$. Então decidiu-se por alterar o número de corpos de 1.000 para 32.768 e executar o software para uma interação apenas, omitindo os cálculos de energia. Com toda a parte de simulação pronta restou estabelecer os mecanismos de comparação entre as várias técnicas. Para tal as funções foram chamadas uma a uma e seus tempos de execução armazenados em variáveis. As funções compiladas tiveram ambos os tempos de execução armazenados em variáveis diferentes para posterior comparação.

Uma vez que todo o código foi finalizado, com a simulação em versões para cada método a ser testado, função para controle e chamada da simulação e variáveis de armazenamento para o tempo de execução de cada método, o algoritmo foi executado em dois ambientes distintos, dois computadores com diferentes configurações de hardware e software, o computador em nuvem do Google Colab e um notebook Dell G3.

O Google Colab é um serviço de desenvolvimento colaborativo em nuvem do Google. Ele fornece interface web com funcionalidades similares ao do software Jupyter

Notebook, permitindo a escrita do código em células e sua execução sob demanda, inserção de texto e formatação Markdown. O serviço aloca parte da memória, disco rígido e acesso à CPU para cada sessão, permitindo ao usuário usar parte dos recursos computacionais da máquina. Também é possível requisitar o uso de GPU alterando o tipo de sessão (*runtime*), ao fazê-lo parte do poder computacional da GPU também estará disponível. O Google Colab já possui vários softwares pré-instalados para uso em programação científica, mas também permite a instalação de outros pacotes para uso na sessão, esses ficam salvos temporariamente até que a execução do kernel da sessão seja terminada.

Esse computador é um servidor que conta com CPU Intel Xeon E5-2699, com clock de 2,3 GHz a 3,6 GHz, 18 núcleos (36 *threads*) e 45MB de cache, GPU dedicada NVidia Tesla K80 24GB 2x2496 núcleos CC3.7. É possível acessar 2 núcleos do processador Xeon, 12GB a 13GB de RAM e 2496 núcelos e 11GB de RAM da GPU K80.

O servidor roda Ubuntu 18.04.1 64 bits, com kernel linux 4.14.79, gcc 7.3.0, Python 3.6.7, Numpy 1.14.6, Numba 0.40.1 e CUDA 9.2.

O notebook Dell G3 possui CPU Intel core i7-8750H com clock de 2,2 GHz a 4,1 GHz, 6 núcleos (12 *threads*) e 9MB de cache, GPU integrada Intel UHD Graphics 630 e GPU dedicada NVIDIA GeForce GTX 1050 Ti Mobile 4GB 768 núcleos CC6.1.

Esse computador roda Manjaro Linux 18 Illyria 64 bits, kernel 4.20.11, gcc 8.2.1, driver NVIDIA 415.25, CUDA 10.0, Python 3.7.1, Numpy 1.15.4, Numba 0.41.0 com CUDAToolkit 9.2 e Anaconda versão 2018-12.

Os arquivos ".ipnb"com o código e os resultados para ambos os computadores estão disponíveis no sítio <https://github.com/FMHemerli/TCCAstro>.

4 Resultados e Discussões

Os dados obtidos dos diferentes métodos em diferentes configurações de hardware estão relacionados abaixo (o símbolo "//"indica que o método utiliza paralelismo.):

Ferramenta	Google Colab (s)	Dell G3 (s)
C	-	7,8844
C OpenMP //	-	1,3900
Python Puro	1.451,5847	648,0228
Numpy	55,9083	31,2096
Numba CPU	11,5298	4,9287
Numpy Numba CPU	33,2135	16,8520
Numba CPU //	6,4286	0,7550
Numpy Numba CPU //	17,2156	14,9583
Numba GPU //	0,2865	0,7350
Numba GPU // kernel	0,2828	0,7333

Tabela 1 – Tempos de execução de cada ferramenta por hardware

Os resultados dessa tabela mostram ganho de desempenho quão mais complexa a aplicação das ferramentas utilizadas, com a adição dos dados de desempenho para simulação em C e C OpenMP (paralelismo em CPU) no computador Dell G3 para efeito comparativo com ferramentas mais populares. Nota-se que o uso de Numba com paralelismo em CPU consegue ser quase duas vezes mais rápido que C OpenMP, e Numba em CPU serial é consideravelmente mais rápido que C.

Estranhamente as combinações de Numpy com Numba em CPU, tanto serial quanto paralelizado, apresentam uma queda de desempenho em relação às ferramentas com uma menor complexidade de implementação. A adição de Numpy através do uso de numpy arrays às ferramentas Numba CPU serial e paralelizado reduzem drasticamente o desempenho, embora o uso de Python Numpy (sem Numba) represente um grande ganho e a despeito da documentação do Numba alegar que a ferramenta tem afinidade com Numpy.

Os resultados esperados para esse casos, no hardware do Google Colab por exemplo,

seriam Numpy Numba CPU apresentar um tempo de execução entre $11,53s$ e $6,43s$. E para Numpy Numba CPU paralelizado um valor entre $6,43s$ e $0,29s$. Para o Dell G3 a queda no uso de paralelismo é ainda mais evidente, usar Numpy com Numba é cerca de vinte vezes mais lento.

	Python	Numpy	Numba CPU	Numba CPU //	GPU
Python	1,0000	0,0385	0,0079	0,0044	0,0002
Numpy	25,9637	1,0000	0,2062	0,1150	0,0051
Numba CPU	125,8982	4,8490	1,0000	0,5576	0,0248
Numba CPU//	225,8027	8,6967	1,7935	1,0000	0,0446
Numba GPU	5.067,5493	195,1785	40,2512	22,4424	1,0000

Tabela 2 – Relação de desempenho entre ferramentas para Google Colab.

Aqui podemos ver as comparações entre os tempos das ferramentas para o computador na nuvem Google Colab. Destaque para o ganho gritante de desempenho do uso de Numba com paralelismo em GPU. A ferramenta que permite comunicar um código Python com a tecnologia CUDA é mais de cinco mil vezes mais rápida que o uso de Python puro, quase duzentas vezes mais rápida que Python Numpy, quarenta vezes mais rápida que Python com Numba serial e mais de vinte vezes mais rápida que o segundo método mais eficiente utilizado.

Diminuindo a complexidade, vemos que o uso de Numba com paralelismo em CPU também resulta num salto grande de desempenho, e essa é uma ferramenta extremamente fácil de implementar, dependendo apenas da correta declaração da função com o decorador. Uma única linha de código que faz o execução ser mais de duzentas vezes mais rápida, sendo uma excelente ferramenta na resolução de problemas que requeiram muito desempenho.

Como o computador Dell G3 possui placa gráfica dedicada inferior a do Google Colab, notamos resultados muito próximos para paralelismo em CPU e GPU. Esse notebook conta com uma GPU voltada para jogos e otimizada para baixo consumo de energia. A similaridade dos dados também se deve ao processador Intel Core i7 presente nesse equipamento ser uma CPU de altíssimo desempenho para computadores domésticos. Nota-se que ao aplicar paralelismo em CPU na execução do algoritmo, o mesmo é executado mais de oitocentas vezes mais rápido que em em

	Python	Numpy	Numba CPU	Numba CPU //	GPU
Python	1,0000	0,0482	0,0076	0,0012	0,0011
Numpy	20,7636	1,0000	0,1579	0,0242	0,0236
Numba CPU	131,4795	6,3322	1,0000	0,1532	0,1491
Numba CPU//	858,3083	41,3372	6,5281	1,0000	0,9735
Numba GPU	881,6637	42,4620	6,7057	1,0272	1,0000

Tabela 3 – Relação de desempenho entre ferramentas para Dell G3

Python puro, reduzindo o tempo gasto de cerca de 11 min para menos de 1s. Esse aumento notável da eficiência do programa utilizando uma técnica que consiste, resumidamente, em adicionar uma linha de código '@jit(parallel=True)' antes da função, demonstra o quanto se pode fazer em termos de computação científica com Python, sem a necessidade de recorrer a linguagens de programação muito mais complexas como C/C++ e FORTRAN.

Para o físico que necessita modelar computacionalmente sistemas complexos e para o aluno de graduação em Física que aprende programação mas não vê uma oportunidade de aplicar esse conhecimento de forma proveitosa na sua formação, os dados levantados por essa pesquisa mostram que existe a possibilidade de, sem a necessidade de despender horas aprendendo uma linguagem de programação mais eficiente e menos versátil, resolver problemas físicos com alto nível de complexidade. Mesmo o Numpy, módulo Python conhecido por aumentar a eficiência desse tipo de algoritmo e amplamente utilizado em diversas áreas, apresenta desempenho muito inferior ao uso de Numba com paralelismo em CPU, tanto no computador do Google Colab quanto no Dell G3, sendo que neste primeiro, o paralelismo em GPU apresenta velocidade de execução mais de cinco mil vezes maior que o uso de Python puro, em comparação com os quase vinte seis de Numpy. Vemos que o ganho, no Google Colab, da técnica usando CUDA é de cerca de 195 vezes sobre o Numpy, e usando paralelismo em CPU é de oito vezes. Enquanto no Dell G3 o ganho de Numba com paralelismo em CPU sobre o Numpy é de 41 vezes, devido ao uso de uma CPU com mais núcleos, mais memória cache e sem as restrições impostas para o uso do Google Colab.

Os dados obtidos indicam que é possível praticar computação científica de alto nível

utilizando Python, a despeito do comumente aceito. Com as ferramentas corretas é possível executar algoritmos em tempo compatível com técnicas mais avançadas de programação, com a vantagem do tempo reduzido para a implementação do código. Sendo Python uma linguagem conhecida por sua simplicidade, com o advento de grandes plataformas de desenvolvimento em Python científico como o SciPy e o Anaconda, o uso de módulos como o Numba, que propiciam aumento notável no desempenho de *scripts* Python, cria a oportunidade para pesquisadores e alunos de adentrar à área da computação científica, um ramo que se torna cada vez mais relevante conforme as tecnologias avançam e os desafios da Física se tornam mais complexos. A possibilidade de se utilizar uma linguagem de programação conhecida por sua curva de aprendizagem amigável na resolução de problemas complexos com desempenho como o demonstrado torna a prática muito convidativa e relevante.

Para efeito demonstrativo, podemos comparar os resultados obtidos para outro cálculo, o cálculo de μ_0 de Cosmologia Observacional, utilizando diversos métodos com paralelismo em CPU e GPU, incluindo Numba:

	Dell G3 (s)	Google Colab (s)
C	0,27300	0,54200
C OpenMP //	0,04740	0,27600
Python	35,60000	59,90000
Python NumPy	1,65000	1,620000
Python Numba	0,35800	0,61900
Python Numpy Numba	0,29500	0,79100
Python Numba //	0,13100	0,60200
Python Numpy Numba //	0,04640	0,41300
CUDA SP	0,00128	0,00156
CUDA DP	0,03770	0,00488
PyCUDA SP	0,00401	0,00594
PyCUDA DP	0,03780	0,00654
Numba GPU SP	0,04930	0,00833
Numba GPU DP	0,06350	0,00693

Tabela 4 – Teste μ_0 para comparação de métodos

Essa tabela apresenta uma visão mais abrangente sobre o desempenho de Python com Numba. Embora ela não tenha relação com o desenvolvimento da simulação

n-body, podemos ver a relação de desempenho que Numba CPU e GPU apresentam comparados a CUDA escrito em C/C++ com declaração complexa de kernel, PyCUDA – uma outra solução Python para paralelismo em GPU, que exige declaração de *kernel* CUDA – e OpenMP que é utilizado para paralelismo em CPU na linguagem C. Embora Numba GPU ainda esteja aquém de um código CUDA escrito em C/C++, se considerarmos que Numba utiliza linguagem Python e não requer a declaração de um *kernel*, ele é uma boa opção para desenvolvedores que necessitam de paralelismo e não dispõem de tempo para aprender programação CUDA.

Deve-se lembrar que a pesquisa utilizou uma simulação do problema de n-corpos gravitacional devido a simplicidade de sua modelagem e grande complexidade de resolução, utilizando o método de aproximação numérica de Euler e a abordagem de força bruta pelos mesmos motivos, preservando a simplicidade da abordagem e agravando a complexidade de resolução. Uma vez que esta última cabe aos computadores, o problema de n-corpos serve o propósito de teste de desempenho do método como aplicação em Física para um caso de grande complexidade. Ele não representa uma medida genérica de desempenho para qualquer aplicação. Apesar disso, os tempos obtidos para cada método indicam que problemas desse tipo se beneficiam muito do paralelismo em CPU e GPU fornecidos pelo módulo Numba. Pode-se supor que uma relação de desempenho similar pode ser obtida para problemas de múltiplas cargas em eletromagnetismo (problema de n-corpos eletromagnético), uma vez que a força elétrica também é uma força de campo conservativa, inversamente proporcional à distância. E mesmo para outras aplicações diferentes esses dados ainda indicam a possibilidade de alto ganho na utilização de Python com Numba.

5 Considerações Finais

A análise dos dados nos mostra que é possível obter bom desempenho utilizando Python para computação científica na resolução de problemas complexos. O módulo Numba torna isso possível ao trazer para a linguagem a possibilidade de compilar funções e executá-las em paralelismo em CPU ou em GPU através de CUDA. Sabendo que Python é uma linguagem de programação conhecida pela sua simplicidade e facilidade de aprendizado, essa conclusão indica que o uso de Python e suas ferramentas, especialmente Numba, são extremamente vantajoso para pesquisa em Física. Sendo assim, o aprendizado de Python nos cursos de graduação em Física se torna mais relevante, pois demonstra que modelar problemas físicos em Python não é meramente uma forma extravagante de os resolver, mas uma poderosa ferramenta de pesquisa. O uso dessa linguagem é difundido entre pesquisadores de várias áreas, estando presente até mesmo na recente imagem do buraco negro da galáxia M87 pela equipe do projeto EHT. Ou seja, Python já é utilizado para pesquisa de ponta, a possibilidade da utilização de computação paralela na linguagem só aumenta as possibilidades de sua utilização. São muitos os motivos para se aprender Python.

A simulação desenvolvida atende a uma necessidade bastante específica, ela não representa todos os casos possíveis de modelagem computacional, podendo até mesmo haver situações em que o uso de Numba seja ruim. São particularidades de cada caso e cada método. Apesar disso os dados obtidos mostram ao menos que para esse tipo de problema o uso de paralelismo é extremamente vantajoso. Alguns métodos não são compatíveis com determinados problemas, mas no que diz respeito a paralelismo ao menos é possível consultar modelos escritos em outras linguagens, e testar. Especificamente problemas que já foram modelados para CUDA em outras linguagens possuem grandes chances de serem compatíveis com Numba quando usado paralelismo em GPU, visto que é a mesma tecnologia. Em suma, a especificidade do teste desenvolvido na pesquisa não diminui seu mérito, simplesmente se limita a um escopo menos geral.

A partir desse ponto, demonstrando a viabilidade e as vantagens da ferramentas,

é possível testar o ganho em outros dispositivos menos convencionais como mini-computadores e microcontroladores. Demonstrada a possibilidade de resolver esse tipo de problema em hardware com maiores limitações, seria possível, por exemplo, montar um laboratório de computação científica utilizando esses dispositivos, permitindo essa prática a um custo muito baixo. Por outro lado, é possível aumentar a complexidade do modelo criado para simular um sistema mais realista, alterando o método de aproximação numérica e a distribuição de cargas.

Numba abriu um enorme leque de possibilidades para Python científico. Resta que seja notado, motivos não faltam para se aprender Python.

Referências

BELLEMAN, R. G.; BÉDORF, J.; ZWART, S. F. P. High performance direct gravitational n-body simulations on graphics processing units ii: An implementation in cuda. *New Astronomy*, Elsevier, v. 13, n. 2, p. 103–112, 2008. Citado na página 10.

BRIGGS, J. *Python for kids: A playful introduction to programming*. 1. ed. [S.l.]: No Starch Press, 2013. Citado na página 9.

COLPO, R. A.; FARIA, A. U. de; MACHADO, A. F. O ensino de física no ensino médio intermediado por programação em linguagem python. *Anais do X ENPEC*, 2015. Citado na página 9.

COSTA, A. C. et al. Python: Será que é possível numa escola pública de ensino médio? In: *Anais do Workshop de Informática na Escola*. [S.l.: s.n.], 2017. v. 23, n. 1, p. 255. Citado na página 9.

DANIEL, J. L.; FOSTER-O'NEAL, J. K. The numerical open-source many-body simulator (noms). 2012. Citado na página 19.

FANGOHR, H. A comparison of c, matlab, and python as teaching languages in engineering. In: SPRINGER. *International Conference on Computational Science*. [S.l.], 2004. p. 1210–1217. Citado na página 9.

GIORDANO, N. J.; NAKANISHI, H. Computational physics. Prentice-Hall, 2006. Citado na página 19.

GOZ, D. et al. Direct n-body code on low-power embedded arm gpus. In: SPRINGER. *Intelligent Computing-Proceedings of the Computing Conference*. [S.l.], 2019. p. 179–193. Citado na página 12.

HALLIDAY, D.; RESNICK, R.; WALKER, J. *Fundamentos de Física: Gravitação, Ondas E Termodinâmica. Vol. 2* . [S.l.]: Grupo Gen-LTC, 2000. Citado 3 vezes nas páginas 11, 14 e 21.

MARQUES, D. L. et al. Atraindo alunos do ensino médio para a computação: Uma experiência prática de introdução à programação utilizando jogos e python. In: *Anais do Workshop de Informática na Escola*. [S.l.: s.n.], 2011. v. 1, n. 1, p. 1138–1147. Citado na página 9.

MEER, G. van der. *How we use Python at Spotify*. 2013. <https://labs.spotify.com/2013/03/20/how-we-use-python-at-spotify/>. Accesso em: 2019-06-30. Citado na página 9.

NUMBA Documentation. 2019. <http://numba.pydata.org/numba-doc/0.38.0/index.html>. Accesso em: 2019-06-30. Citado 3 vezes nas páginas 12, 21 e 26.

NVIDIA cuda c programming guide. 2019. <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>. Accesso em: 2019-06-30. Citado 2 vezes nas páginas 10 e 26.

NYLAND, L.; HARRIS, M.; PRINS, J. *Fast n-body simulation with cuda*. 2007. Citado 3 vezes nas páginas 16, 18 e 21.

PARECER CNE/CES 1.304/2001 - Diretrizes Nacionais Curriculares para os Cursos de Física. 2001. <http://portal.mec.gov.br/cne/arquivos/pdf/CES1304.pdf>. Accessed: 2019-06-30. Citado na página 9.

PERKEL, J. M. Why jupyter is data scientists' computational notebook of choice. *Nature*, v. 563, n. 7729, p. 145, 2018. Citado na página 20.

PYTHON at Google: Python as an official language at Google. Python usage on Google various applications. 2012. <https://quintagroup.com/cms/python/google>. Accesso em: 2019-06-30. Citado na página 9.

RAPOPORT, R. M. B. C. J. B. C. *Python at Netflix*. 2013. <https://medium.com/netflix-techblog/python-at-netflix-86b6028b3b3e>. Accesso em: 2019-06-30. Citado na página 9.

ROSSUM, G. van. *Python tutorial Kernel Description*. 1995. <www.python.org>. Accesso em: 2019-06-30. Citado na página 9.

WEB Service Efficiency at Instagram with Python. 2016. <https://instagram-engineering.com/web-service-efficiency-at-instagram-with-python-4976d078e366>. Accesso em: 2019-06-30. Citado na página 9.

A Código-fonte *n-body* Gravitacional

```
# Bibliotecas importadas
import math
import numpy as np
from timeit import default_timer as timer
import numba as nb
from numba import jit, njit, prange, cuda

#Definições
G = 6.67408*1e-11 #define a constante gravitacional
sft = 1e-10 #constante de atenuamento para tratamento das singularidades

#------ Funções de simulação ------#

## Função para Python puro e Numba CPU serial
@njit #decorador do compilador Numba
def gravitationalNbody_NumbaCPU(r, v, a, m, nobj, ninter, dt,
                                flag_Energy=False):
    for inter in range(ninter):
        for i in range(nobj):
            a[i][0] = 0.0; a[i][1] = 0.0; a[i][2] = 0.0
            for j in range(nobj):
                if i != j:
                    # calculate a_ij acceleration
                    dx = r[j][0] - r[i][0]
                    dy = r[j][1] - r[i][1]
                    dz = r[j][2] - r[i][2]
                    dsq = dx*dx + dy*dy + dz*dz + sft*sft
                    gA = (G*m[j])/(dsq*math.sqrt(dsq))
                    # sum a_ij_x to a_i_x, etc
                    a[i][0] += gA*dx
                    a[i][1] += gA*dy
                    a[i][2] += gA*dz
            # update the v_i velocity
            v[i][0] += a[i][0]*dt
            v[i][1] += a[i][1]*dt
```

```
            v[i][2] += a[i][2]*dt
        for i in range(nobj):
            # update the r_i position
            r[i][0] += v[i][0]*dt
            r[i][1] += v[i][1]*dt
            r[i][2] += v[i][2]*dt
    K = 0.0; U = 0.0
    if flag_Energy:
        for i in range(nobj):
            # Sum the total kinectic energy
            K += 0.5*m[i]*(v[i][0]**2 + v[i][1]**2 + v[i][2]**2)
            for j in range(nobj):
                if (i < j):
                    # Sum the total potential energy
                    U += (-G*m[i]*m[j])/math.sqrt((r[j][0] - r[i][0])**2 +
                                                  (r[j][1] - r[i][1])**2 +
                                                  (r[j][2] - r[i][2])**2 +
                                                  sft**2)
    return K, U


## Função para Python com Numba CPU em paralelo
@njit(parallel=True) #decorador do compilador Numba para paralelização em CPU
def gravitationalNbody_NumbaCPUparallel(r, v, a, m, nobj, ninter, dt,
                                        flag_Energy=False):
    for inter in range(ninter):
        for i in prange(nobj):
            a[i][0] = 0.0; a[i][1] = 0.0; a[i][2] = 0.0
            for j in range(nobj):
                if i != j:
                    # calculate a_ij acceleration
                    dx = r[j][0] - r[i][0]
                    dy = r[j][1] - r[i][1]
                    dz = r[j][2] - r[i][2]
                    dsq = dx*dx + dy*dy + dz*dz + sft*sft
                    gA = (G*m[j])/(dsq*math.sqrt(dsq))
                    # sum a_ij_x to a_i_x, etc
                    a[i][0] += gA*dx
                    a[i][1] += gA*dy
```

```
                    a[i][2] += gA*dz
            # update the v_i velocity
            v[i][0] += a[i][0]*dt
            v[i][1] += a[i][1]*dt
            v[i][2] += a[i][2]*dt
        for i in prange(nobj):
            # update the r_i position
            r[i][0] += v[i][0]*dt
            r[i][1] += v[i][1]*dt
            r[i][2] += v[i][2]*dt
    K = 0.0; U = 0.0
    if flag_Energy:
        for i in prange(nobj):
            # Sum the total kinectic energy
            K += 0.5*m[i]*(v[i][0]**2 + v[i][1]**2 + v[i][2]**2)
            for j in range(nobj):
                if (i < j):
                    # Sum the total potential energy
                    U += (-G*m[i]*m[j])/math.sqrt((r[j][0] - r[i][0])**2 +
                                                  (r[j][1] - r[i][1])**2 +
                                                  (r[j][2] - r[i][2])**2 +
                                                  sft**2)
    return K, U


## Função para Python com Numpy em CPU serial
@njit #decorador do compilador Numba
def gravitationalNbody_NumPy_NumbaCPU(r, v, a, m, nobj, ninter, dt,
                                      flag_Energy=False):
    for inter in range(ninter):
        # zero the a acceleration
        a = np.zeros((nobj, 3))
        for i in range(nobj):
            # update the a acceleration
            dr = r[i] - r
            dsq = np.sum(dr*dr, axis=1) + sft*sft
            gA = (G*m[i])/(dsq*np.sqrt(dsq))
            a += (dr.T*gA).T
        # update the v velocity
```

```
        v += a*dt
        # update the r position
        r += v*dt
    K = 0.0; U = 0.0
    if flag_Energy:
        # Sum the total kinetic energy
        K = np.sum(0.5*m*np.sum(v*v, axis=1))
        U = 0.0
        for i in range(nobj):
            for j in range(nobj):
                if (i < j):
                    # Sum the total potential energy
                    U += (-G*m[i]*m[j])/math.sqrt((r[j][0] - r[i][0])**2 +
                                                  (r[j][1] - r[i][1])**2 +
                                                  (r[j][2] - r[i][2])**2 +
                                                  sft**2)
    return K, U

## Função para Python com Numpy e Numba CPU em paralelo
@njit(parallel=True) #decorador do compilador Numba para paralelização em CPU
def gravitationalNbody_NumPy_NumbaCPUparallel(r, v, a, m, nobj, ninter, dt,
                                              flag_Energy=False):
    for inter in range(ninter):
        # zero the a acceleration
        a = np.zeros((nobj, 3))
        for i in prange(nobj):
            # update the a acceleration
            dr = r[i] - r
            dsq = np.sum(dr*dr, axis=1) + sft*sft
            gA = (G*m[i])/(dsq*np.sqrt(dsq))
            a += (dr.T*gA).T
        # update the v velocity
        v += a*dt
        # update the r position
        r += v*dt
    K = 0.0; U = 0.0
    if flag_Energy:
        K = np.sum(0.5*m*np.sum(v*v, axis=1))
        U = 0.0
```

```
        for i in prange(nobj):
            for j in range(nobj):
                if (i < j):
                    # Sum the total potential energy
                    U += (-G*m[i]*m[j])/math.sqrt((r[j][0] - r[i][0])**2 +
                                                  (r[j][1] - r[i][1])**2 +
                                                  (r[j][2] - r[i][2])**2 +
                                                  sft**2)
    return K, U

### Funções para Python com Numba paralelizado em GPU via CUDA
# Cálculo da Aceleração e atualização das velocidades
@cuda.jit("(float64[:,:], float64[:,:], float64[:,:], float64[:], int32, float64)")
#cuda.jit é o decorador do compilador Numba para paralelismo em GPU
def gravitationalNbody_a_v_ij_NumbaGPU64(r, v, a, m, nobj, dt):
    i = cuda.grid(1)
    if i < nobj:
        a[i][0] = 0.0; a[i][1] = 0.0; a[i][2] = 0.0
        for j in range(nobj):
            if i != j:
                dx = r[j][0] - r[i][0]
                dy = r[j][1] - r[i][1]
                dz = r[j][2] - r[i][2]
                dsq = dx*dx + dy*dy + dz*dz + sft*sft
                gA = (G*m[j])/(dsq*math.sqrt(dsq))
                # sum a_ij_x to a_i_x, etc
                a[i][0] += gA*dx
                a[i][1] += gA*dy
                a[i][2] += gA*dz
        v[i][0] += a[i][0]*dt
        v[i][1] += a[i][1]*dt
        v[i][2] += a[i][2]*dt

# Atualização das posições
@cuda.jit("(float64[:,:], float64[:,:], int32, float64)")
def gravitationalNbody_r_i_NumbaGPU64(r, v, nobj, dt):
    i = cuda.grid(1)
    if i < nobj:
        r[i][0] += v[i][0]*dt
```

```
        r[i][1] += v[i][1]*dt
        r[i][2] += v[i][2]*dt

# Cálculo da Energia total
@cuda.jit("(float64[:,:], float64[:,:], float64[:], int32, float64[:])")
def gravitationalNbody_K_U_NumbaCUDA(r, v, m, nobj, K_U):
    i = cuda.grid(1)
    if i < nobj:
        K_U[0] += 0.5*m[i]*(v[i][0]**2 + v[i][1]**2 + v[i][2]**2)
        for j in range(nobj):
            if (i < j):
                K_U[1] += (-G*m[i]*m[j])/math.sqrt((r[j][0] - r[i][0])**2 +
                                                   (r[j][1] - r[i][1])**2 +
                                                   (r[j][2] - r[i][2])**2 +
                                                   sft**2)

# Chamada das funções de paralelismo em GPU
@jit(parallel=True)
def gravitationalNbody_NumbaGPU64(r, v, a, m, nobj, ninter, dt, flag_Energy=False):
    threadsperblock = 128
    blockspergrid = (nobj + (threadsperblock - 1)) // threadsperblock
    print("Threads per block = {}; Blocks per grid = {}".format(threadsperblock,
                                                               blockspergrid))
    htod_ti = timer()
    d_r = cuda.to_device(r)
    d_v = cuda.to_device(v)
    d_a = cuda.to_device(a)
    d_m = cuda.to_device(m)
    htod_tf = timer()
    for inter in range(ninter):
        gravitationalNbody_a_v_ij_NumbaGPU64[blockspergrid,
                                             threadsperblock](d_r, d_v, d_a,
                                                              d_m, nobj, dt)
        cuda.synchronize()
        gravitationalNbody_r_i_NumbaGPU64[blockspergrid,
                                          threadsperblock](d_r, d_v, nobj, dt)
        cuda.synchronize()
    dtoh_ti = timer()
    d_r.copy_to_host(r)
```

```
    d_v.copy_to_host(v)
    d_a.copy_to_host(a)
    d_m.copy_to_host(m)
    dtoh_tf = timer()
    K = 0.0; U = 0.0
    if flag_Energy:
        for i in prange(nobj):
            # Sum the total kinectic energy
            K += 0.5*m[i]*(v[i][0]**2 + v[i][1]**2 + v[i][2]**2)
            for j in range(nobj):
                if (i < j):
                    # Sum the total potential energy
                    U += (-G*m[i]*m[j])/math.sqrt((r[j][0] - r[i][0])**2 +
                                                  (r[j][1] - r[i][1])**2 +
                                                  (r[j][2] - r[i][2])**2 +
                                                  sft**2)
    print("Copied data from CPU to GPU after {} s".format(htod_tf - htod_ti))
    print("GPU calculations done after {} s".format(dtoh_ti - htod_tf))
    print("Copied data from GPU to CPU after {} s".format(dtoh_tf - dtoh_ti))
    print("Copies CPU<->GPU and GPU calculations after {} s".format(dtoh_tf -
                                                                    htod_ti))
    return K, U
    Energies_K_U = np.zeros(2)
    if flag_Energy:
        gravitationalNbody_K_U_NumbaCUDA[blockspergrid,
threadsperblock](r, v, m, nobj, Energies_K_U)
        cuda.synchronize()
    return Energies_K_U[0], Energies_K_U[1]

# Função de controle para uniformização das chamadas das funções de simulação #

# n_cube_edge : número de pontos l de cada aresta de um cubo
# ninter : M, número de interações
# dt : dt, intervalo entre interações
def run_nbody(gravitationalNbodyfunction, n_cube_edge=10, ninter = 5, dt = 0.01,
              flag_array=True, flag_compile=True, flag_Energy= False,
              flag_verbose=False):
    nbody_version = "1.6.3"
```

```
nobj = n_cube_edge**3 # define o número de objetos na simulação
if flag_array:
    # inicializa uma lista de N posições 3D, nulas inicialmente
    positions = np.zeros((nobj, 3))
    # inicializa uma lista de N velocidades 3D, nulas inicialmente
    velocities = np.zeros_like(positions)
    # inicializa uma lista de N acelerações 3D, nulas inicialmente
    accelerations = np.zeros_like(positions)
    # inicializa uma lista de N massas, aqui iguais a 10^6 kg
    masses = 1e9*np.ones(nobj)
else:
    # inicializa uma lista de N de objetos para posição e massa, nulas
    # inicialmente
    positions = [[0., 0., 0.] for i in range(nobj)]
    # inicializa uma lista de N velocidades 3D, nulas inicialmente
    velocities = [[0., 0., 0.] for i in range(nobj)]
    # inicializa uma lista de N acelerações 3D, nulas inicialmente
    accelerations = [[0., 0., 0.] for i in range(nobj)]
    # inicializa uma lista de N massas, aqui iguais a 10^6 kg
    masses = [1e9]*nobj
# Cria um cubo de l^3 pontos, onde cada aresta tem l pontos
scale = 1.0  # escala do cubo, com comprimento da aresta = scale*n_cube_edge
n_half_cube_edge = int(n_cube_edge/2)
for i in range(n_cube_edge):
    for j in range(n_cube_edge):
        for k in range(n_cube_edge):
            positions[k + j*n_cube_edge + i*n_cube_edge*n_cube_edge] = [
                (i - n_half_cube_edge)*scale,
                (j - n_half_cube_edge)*scale,
                (k - n_half_cube_edge)*scale]
if flag_compile:
    # To Numba compile the function 1st time before calling with full data
    positions_backup = positions.copy()
    velocities_backup = velocities.copy()
    accelerations_backup = accelerations.copy()
    print("N-Body (C) Flavio Manoel, v{}".format(nbody_version))
    print("N = {} corpos, dt = {}s, M = {} interações".format(nobj, dt,
                                                              ninter))
    start = timer()
```

```
        Kf, Uf = gravitationalNbodyfunction(positions, velocities,
                                            accelerations, masses, nobj, ninter,
                                            dt, flag_Energy=flag_Energy)
        end = timer()
        if flag_Energy:
            print("Após {} passos, {:.4f}s, Ef = {}, Kf = {}, Uf =
                  {}".format(ninter, ninter*dt, Kf + Uf, Kf, Uf))
        dt_compile_run = end - start
        print("Código executado em {:.6f} s\n".format(dt_compile_run))
        positions = positions_backup.copy()
        velocities = velocities_backup.copy()
        accelerations = accelerations_backup.copy()
    else:
        dt_compile_run = 0.0

    print("N-Body (C) Flavio Manoel, v{}".format(nbody_version))
    print("N = {} corpos, dt = {}s, M = {} interações".format(nobj, dt, ninter))
    if flag_verbose:
        print("Posições = {}\nVelocidades = {}\nAcelerações = {}\nMassas = {}
              \n".format(positions,
              velocities, accelerations, masses))
    start = timer()
    if flag_Energy:
        Ki, Ui = gravitationalNbodyfunction(positions, velocities,
                                            accelerations, masses, nobj, 0, dt,
                                            flag_Energy=flag_Energy)
        print("Ei = {}, Ki = {}, Ui = {}".format(Ki + Ui, Ki, Ui))
    Kf, Uf = gravitationalNbodyfunction(positions, velocities, accelerations,
                                        masses, nobj, ninter, dt,
                                        flag_Energy=flag_Energy)
    end = timer()
    if flag_Energy:
        erro_perc = 100.0*(Kf + Uf - (Ki + Ui))/(Ki + Ui)
        print("Após {} passos, {:.4f}s :\nEf = {}, Kf = {}, Uf = {},
              erro em E = {}%".format(ninter,
                                      ninter*dt, Kf + Uf, Kf, Uf, erro_perc))
    else:
        erro_perc = 0.0
    dt_run = end - start
```

```
    print("Código executado em {:.6f} s".format(dt_run))
    if flag_verbose:
        print("\nPositions = {}\nVelocities = {}\nAccelerations =
              {}".format(positions, velocities, accelerations))
    return dt_compile_run, dt_run, erro_perc

#------ Definições de constantes da Simulação ------#
n_cube_edge=32; ninter = 1; dt = 0.01

#------ Chamadas das funções de teste ------#

## Teste de nbody em Python puro
t_all_py, t_py, erro_py = run_nbody(gravitationalNbody_NumbaCPU.py_func,
                                    n_cube_edge=n_cube_edge, ninter=ninter,
                                    dt=dt, flag_array=False, flag_compile=False)

## Teste de nbody em Python NumPy
t_all_np, t_np, erro_np = run_nbody(gravitationalNbody_NumPy_NumbaCPU.py_func,
                                    n_cube_edge=n_cube_edge,
                                    ninter=ninter, dt=dt, flag_compile=False)

## Teste de nbody em Python Numba CPU
t_all_nb, t_nb, erro_nb = run_nbody(gravitationalNbody_NumbaCPU,
                                    n_cube_edge=n_cube_edge, ninter=ninter,
                                    dt=dt)

## Teste de nbody em NumPy Numba CPU
t_all_np_nb, t_np_nb, erro_np_nb = run_nbody(gravitationalNbody_NumPy_NumbaCPU,
                                             n_cube_edge=n_cube_edge,
                                             ninter=ninter, dt=dt)

## Teste de nbody em Python Numba CPU //
t_all_nb_parallel,
t_nb_parallel,
erro_nb_parallel = run_nbody(gravitationalNbody_NumbaCPUparallel,
                             n_cube_edge=n_cube_edge, ninter=ninter, dt=dt)

## Teste de nbody em NumPy Numba CPU //
t_all_np_nb_parallel,
```

```
t_np_nb_parallel,
erro_np_nb_parallel = run_nbody(gravitationalNbody_NumPy_NumbaCPUparallel,
                                n_cube_edge=n_cube_edge, ninter=ninter, dt=dt)

## Teste de nbody em Python Numba GPU
t_all_gpu,
t_gpu,
erro_gpu = run_nbody(gravitationalNbody_NumbaGPU64, n_cube_edge=n_cube_edge,
                     ninter=ninter, dt=dt)
```