Melhorando um Simulador Utilizando Paralelismo de GPU e Threading em Python

Celio Ludwig Slomp
Instituto Federal Catarinense
Timbó, SC, Brasil
slompcelio132@gmail.com

ABSTRACT

This paper aims to present two parallelism techniques applied to a collision simulator, with the goal of improving performance and allowing the addition of more circles to the simulation. The techniques explored are the use of multi-threading with the threading library and GPU acceleration with the CuPy library. The implementation of CuPy was carried out by replacing mathematical operations with the library's specific functions, while in the case of multi-threading, a function was created to draw the bodies on the screen and perform collision checks. The use of GPU resulted in unsatisfactory performance, with an average of 15 FPS for only two circles, due to the high time required to initialize the CUDA kernel. On the other hand, multi-threading provided a significant improvement, achieving an average of 95 FPS, but it failed to draw all the bodies correctly due to Python's interpreter limitation, which prevents the simultaneous use of multiple cores. These results highlight the importance of choosing the right technologies for high-performance projects.

CCS CONCEPTS

• **High Performance Computing** → Parallel Programming → Threads and GPU.

KEYWORDS

Python, CuPy, Threading, GPU Programming.

1 Introdução

Para a área da ciência da computação, um dos pilares para uma melhor performance de execução é a utilização de paralelismo. O paralelismo é um tipo de computação que se baseia em utilizar dois ou mais núcleos de processamento (ou CPU) para executar processos de um programa simultaneamente.

Sabendo disso, foi escolhido um simulador desenvolvido em uma disciplina anterior para fazer as possíveis melhorias. Como o programa foi escrito na linguagem de programação Python, as melhorias escolhidas foram as de paralelismo utilizando a GPU com a biblioteca CuPy e o uso de *multi-threading* com a biblioteca threading.

Para realizar essas melhorias, as modificações foram implementadas utilizando as bibliotecas e logo em seguida foi testado o código já melhorado, os resultados foram justificados baseando-se em pesquisas.

2 Metodologia

Para este simulador ser melhorado, duas abordagens foram escolhidas: utilização de paralelismo com *threads* e GPU. Primeiramente será aplicado a modificação no código fonte do programa, realizando as possíveis atualizações. Após a modificação no código ser concluída, a execução deste será realizada.

O computador utilizado para os testes possui o *hardware*:

- Processador: Intel i5-9400f 6 núcleos e 6 *threads*;
- Memória: 16GB;
- GPU: GeForce RTX 2060 6GB.

Para a aceitação dessas melhorias, será analisado em um primeiro momento a quantidade padrão de 100 círculos executando com uma média superior ou igual a 60 FPS. Caso essa média seja superior a 60, será aumentado a quantidade de círculos até ela ser de 60 FPS. Caso seja inferior, será reduzida a quantidade de círculos até a média ser de 60 FPS. Caso a média seja inferior ou igual a metade disso, será considerado como uma implementação impossível. A média de 60 FPS foi escolhida por conta da frequência média dos monitores, que é de aproximadamente 60Hz.

Os códigos foram colocados dentro do repositório do autor no Github acessível por [7].

3 Desenvolvimento

Para tentar melhorar o desempenho desse simulador foram tomadas duas abordagens: utilizar *threads* com a biblioteca threading e a GPU com a biblioteca CuPy[1].

3.1 O Simulador Sem as Melhorias

O simulador de colisões é um programa que simula uma quantidade *N* de círculos se movimentando com a gravidade e se colidindo uns com os outros. A Figura 1 é uma captura de tela deste simulador após alguns segundos de execução.

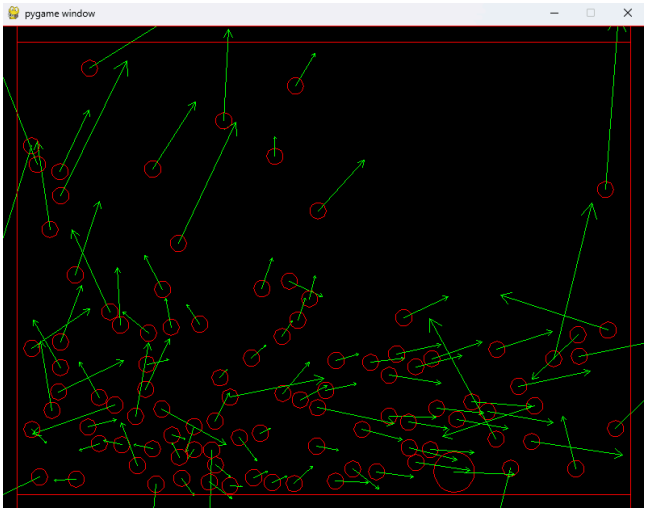

Figura 1: Simulador de Colisões. Fonte: o autor.

Para este *hardware*, o limite do simulador para executar em uma média de 60 FPS é de aproximadamente 100 círculos. Com 180 círculos, a média é de 32 FPS, abaixo da de atualização do monitor.

3.2 Utilização da Biblioteca CuPy

Como a biblioteca CuPy é uma biblioteca matemática, foi escolhido utilizá-la onde operações matemáticas são altamente utilizadas. O arquivo que mais possui operações matemáticas é o *vec2.py*, nele estão todos os cálculos de colisão, verificação, etc.
Para adicionar as funções desta biblioteca ao código bastou remover o operador e então chamar a função desejada com os valores como parâmetro.
Exemplo: A operação "*x* + *y*" passou a ser "*cupy.add*(*x*,*y*)".

3.3 Utilização da Biblioteca Threading

A biblioteca threading executa funções que não possuem retorno e também não realiza operações matemáticas de forma isolada, diferenciando do CuPy. Outro fator relevante para a implementação é o fato de estar sendo utilizada a biblioteca Pygame, que de acordo com os fóruns [2][3][4], não é uma biblioteca *thread safe*. Isso significa que é muito importante deixar a execução desta dentro da *thread* principal do programa.
Sabendo disso, o uso de threading foi implementado com o Pygame, ignorando o fator de *thread safe* e também apenas verificando colisões, tornando o programa totalmente *thread safe*.
Para a implementação *thread safe*, foi criado uma *thread* que executa o método "*loop_bodies*". Este método é responsável por fazer a verificação de colisões entre os corpos da simulação (outros círculos e paredes).
Para a implementação sem ser *thread safe*, foi colocado o laço de repetição que desenha os corpos dentro do método "*loop_bodies*", sendo executado em outra thread, tirando essa parte do Pygame da thread principal do programa.

4 Resultados e Discussões

4.1 Resultados da Biblioteca CuPy

Após a devida modificação no código fonte do programa, foi realizada a execução deste. Em um primeiro momento, a execução foi realizada com 100 círculos, a qual obteve uma média de 0.16 FPS. Após isso, foi realizada a execução com apenas 2 círculos para tentar obter um resultado satisfatório, entretanto, essa simulação teve uma aproximação de 15 FPS, continuando a ser impossível de implementar. Tendo esse péssimo resultado, foi realizada uma pesquisa para tentar achar o motivo deste resultado. Segundo [1], o CuPy precisa iniciar o CUDA *kernel* toda vez que é realizado alguma operação, tornando-o lento para matrizes pequenas.

Size	NumPy [ms]	CuPy [ms]
10^4	0.03	0.58
10^5	0.20	0.97
10^6	2.00	1.84
10^7	55.55	12.48
10^8	517.17	84.73

Figura 2: Comparação do NumPy com o CuPy. Disponível em [1]

Na Figura 2 é possível visualizar a diferença de tempo pelo tamanho das matrizes, o NumPy é muito melhor para uma

[1] Site official da biblioteca CuPy: https://cupy.dev/

matriz com 10^4 elementos, enquanto que o CuPy é muito melhor com uma matriz de 10^8 elementos. Este teste foi realizado pelo desenvolvedor da biblioteca.
Os testes da Figura 2 foram realizados usando o *hardware*:

- Processador: 2 Intel Xeon CPU E5-2630 v3;
- Memória: 64GB
- GPU: 4 GeForce GTX TITAN X

4.2 Resultados da Biblioteca Threading

4.2.1 Implementação thread safe. O resultado com esta abordagem foi um tanto quanto surpreendente, afinal, os resultados da simulação tiveram uma performance minimamente pior que a do programa sem nenhuma melhoria. Ele obteve este resultado por causa do GIL (*Global Interpreter Lock*) do Python, que limita o interpretador do dele para utilizar apenas um núcleo de processamento por vez, mesmo se o CPU do usuário tiver vários núcleos. Além disso, o GIL também pode priorizar as *threads I/O-bound* do que as CPU-*bound*, tornando a velocidade de execução do código inferior ou igual ao do mesmo sem as melhorias.

4.2.2 Implementação ignorando o thread safe. Esta abordagem foi a única que obteve performance real. Para 100 círculos, a simulação estava sendo executada em uma média de 95 FPS e com 140 círculos ficando com uma média de 65 FPS. Observação: com mais de 100 círculos, erros de "*index out of range*" começam a aparecer no terminal. Na Figura 3 pode-se entender o motivo desse aumento de performance.

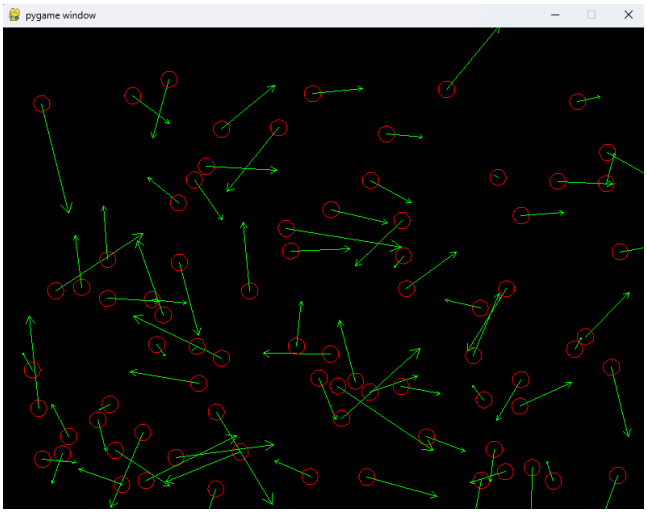

Figura 3: Simulação sem ser *thread safe*. Fonte: o autor.

Esta "melhora" aconteceu por causa do loop principal. Quando o loop é iniciado, o "*self.time_handles.ticks*" obtém os ticks da execução para depois tornar possível o cálculo do FPS. Após isso, o Pygame pinta o fundo da tela de preto e, logo em seguida, a *thread* com o método "*loop_bodies*" é iniciada. Quando o loop reinicia, a tela é pintada de preto antes da *thread* iniciada no loop anterior ter sido finalizada, gerando uma simulação incompleta.

```
def start_main_loop(self):
    while self.running:
        self.time_handler.ticks = pg.time.get_ticks()

        self.screen.fill((0, 0, 0)) # pinta o fundo preto

        # Cria e inicia a thread
        th1 = threading.Thread(target=self.loop_bodies)
        th1.start()

        self.handle_events() # Trata as entradas

        pg.display.flip()

        self.time_handler.wait() # Gera o FPS
```

5 Conclusão

Foi possível observar neste trabalho a implementação de técnicas de paralelismo em um simulador de colisões, com ambos os resultados sendo completamente diferente do esperado. Com a biblioteca threading houve uma melhora no desempenho, sendo possível de adicionar mais círculos, mas ficando uma simulação incompleta por conta do interpretador do Python ter a limitação de usar apenas um núcleo de processamento e o Pygame não conseguir desenhar todos os corpos. Já com a biblioteca CuPy, houve uma piora significativa, pois a inicialização do CUDA era muito custosa em tempo para o tamanho das operações.
Essas melhorias podem ser implementadas em outras aplicações (como jogos e simulações físicas) que necessitem de muito poder computacional. Futuras melhorias podem tentar outras tecnologias como linguagem de programação e bibliotecas, aplicando outras técnicas de programação.
Com isso tudo, a escolha das tecnologias têm um enorme peso para aplicação de técnicas de paralelismo.

REFERENCES

[1] Nishino, R., and Loomis, S. H. C. CuPy: A NumPy-compatible library.
[2] GitHub. (2020). Issue #2806: Pygame in multiple threads. Disponível em: https://github.com/pygame/pygame/issues/2806. Acesso em: 4 dez. 2024.

[3] Stack Overflow. (2010). Pygame in a thread. Disponível em: https://stackoverflow.com/questions/2970612/pygame-in-a-thread. Acesso em: 4 dez. 2024.
[4] Reddit. (2022). How to parallelize in Pygame?. Disponível em: https://www.reddit.com/r/pygame/comments/yhtypv/how_to_parallelize_in_pygame/. Acesso em: 4 dez. 2024.
[5] Python Wiki. Global Interpreter Lock (GIL). Disponível em: https://wiki.python.org/moin/GlobalInterpreterLock. Acesso em: 4 dez. 2024.
[6] Real Python. (2021). What Is the Python Global Interpreter Lock (GIL)?. Disponível em: https://realpython.com/python-gil/. Acesso em: 4 dez. 2024.
[7] Slomp, C.. Programação de Alto Desempenho - Trabalho Final. Disponível em: https://github.com/CelioSlomp/Faculdade/tree/main/ProgramacaoAltoDesempenho/TrabalhoFinalPAD. Acesso em: 4 dez. 2024.