

گلبو رشیدی 810100148

پروا شریفی 810100171

کدها و عکسهای خروجی بخش اول و دوم موجود در:

[https://drive.google.com/drive/folders/121iAFnekA13t\\_En5Yl8Sf3tPqNPgmO-5?usp=sharing](https://drive.google.com/drive/folders/121iAFnekA13t_En5Yl8Sf3tPqNPgmO-5?usp=sharing)

## بخش اول

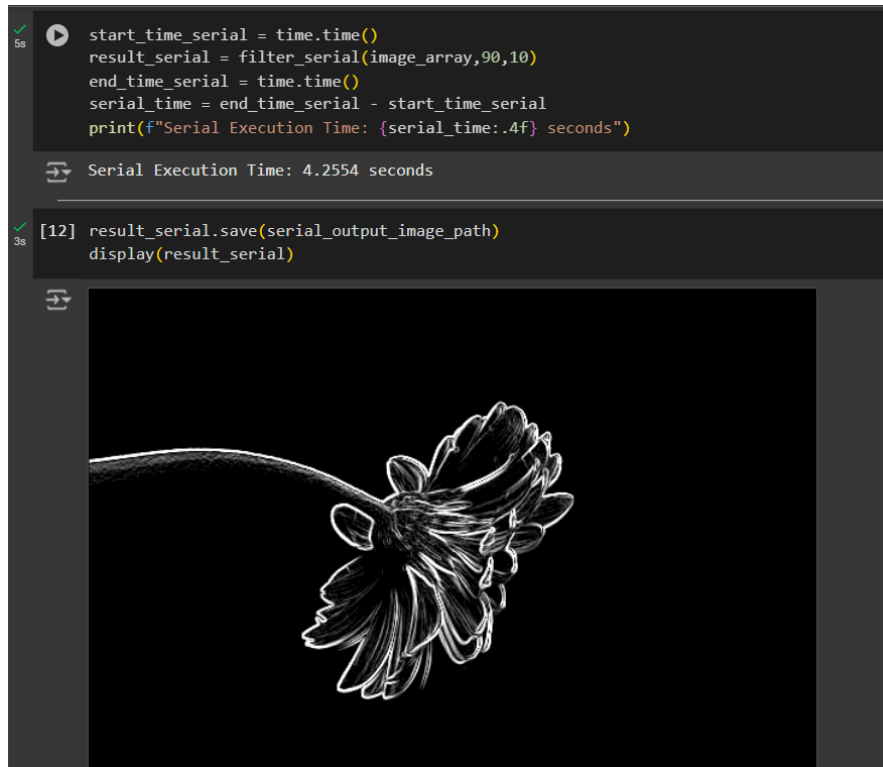
کرنل مورد استفاده (sobel)

```
kernel_x = np.array([[ -1,  0,  1], [-2,  0,  2], [-1,  0,  1]])  
kernel_y = np.array([[ 1,  2,  1], [ 0,  0,  0], [-1, -2, -1]])
```

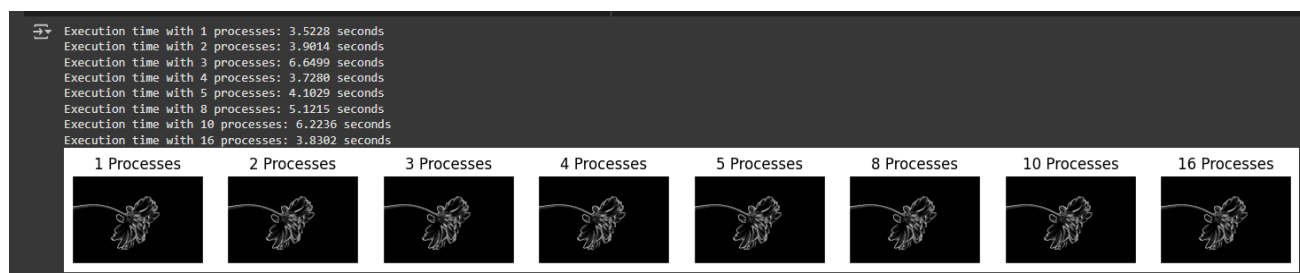
یا حتی:

```
#kernel_x = np.array([[ -3,  0,  3], [-10,  0, 10], [-3,  0,  3]])  
#kernel_y = np.array([[ -3, -10, -3], [ 0,  0,  0], [ 3, 10,  3]])
```

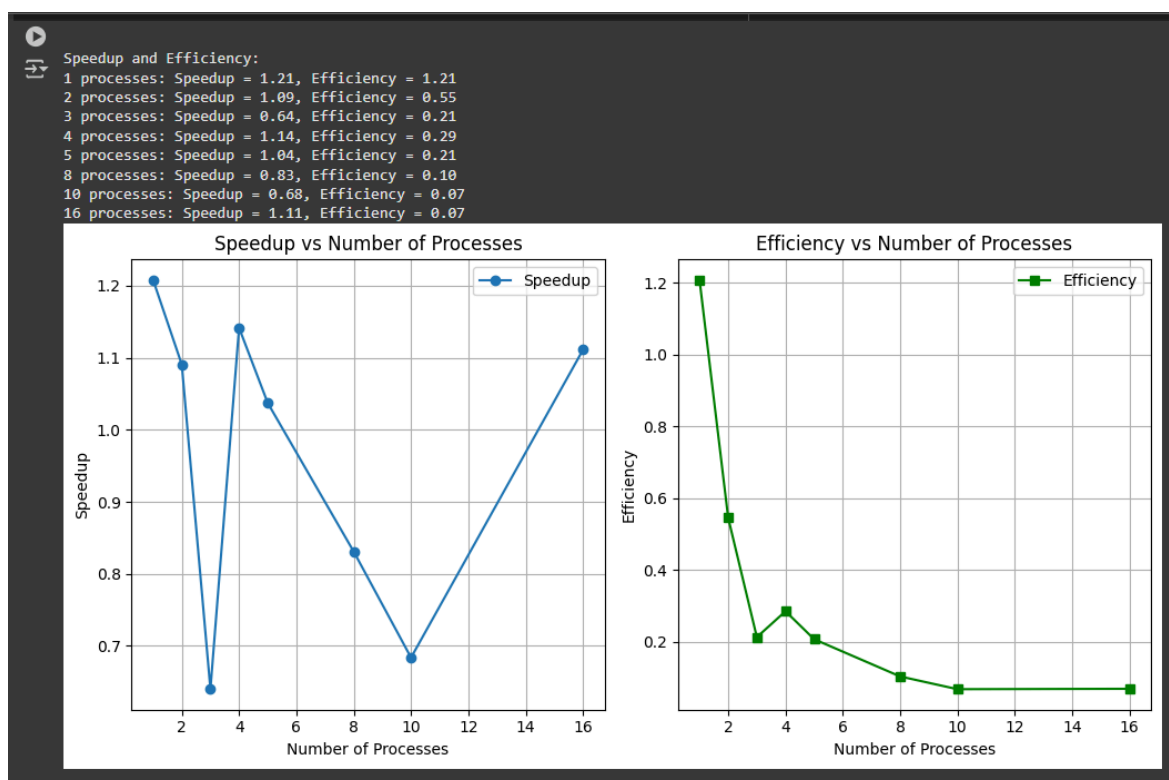
نتیجه کد سریال:



## نتیجه کد موازی:



## آنالیز نتایج:



همانطور که مشاهده میشود، با افزایش تعداد پردازنده‌ها، زمان اجرا گاهی کاهش می‌یابد اما به دلیل سرشار ایجاد پردازنده‌ها و هماهنگی بین آن‌ها، کاهش زمان اجرا از حد مشخصی بهینه نخواهد بود و efficiency پایین است.

## بخش دوم

در ابتدا کد سریال ++C را اجرا می‌کنیم و زمان اجرا را به دست می‌آوریم:


```
Overwriting cpp_serial_version.cpp

[124] !g++ cpp_serial_version.cpp -o cpp `pkg-config --cflags --libs opencv4`
4s

[125] !./cpp
1s

Serial Execution Time: 0.061605 seconds

[128] from PIL import Image
output_image = Image.open("/content/drive/MyDrive/ParallelProgramming/CA/output/output_cpp_serial_image.jpg")
display(output_image)
1s
```



همانطور که مشاهده می‌شود زمان اجرا 0.061605 ثانیه است. پیاده سازی کد موازی با استفاده از کودا را در چند مرحله انجام می‌دهیم و نتیجه را ارزیابی می‌کنیم. ابتدا تعداد تردهای داخل یک بلوک و تعداد بلوک های داخل یک گرید را مشخص می‌کنیم. همچنین مموری های لازم را allocate می‌کنیم و بعدا نیز آن ها را آزاد می‌کنیم. از cudaMemcpy برای انتقال داده بین host و کرنل استفاده می‌کنیم. یک تابع کرنل به اسم sobel\_filter داریم. همچنین برای error handling تمام فراخوانی های توابع کودا را با استفاده از ماکرو CHECK انجام می‌دهیم.(مطابق اسلایدهای درس)

1. اعمال No divergence

لینک های کمک کرده در این بخش:

<https://forums.developer.nvidia.com/t/avoiding-thread-divergence/35975>

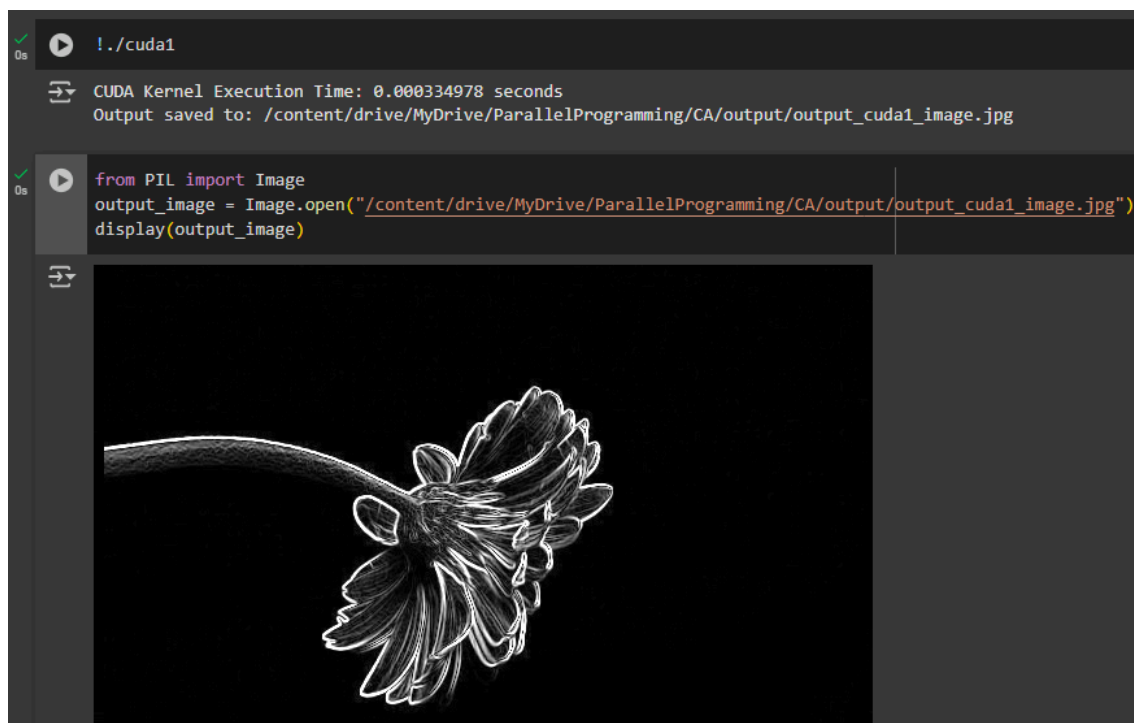
<https://stackoverflow.com/questions/26316317/how-to-avoid-divergent-branch-in-simple-if-statements-in-cuda>

در ابتدا برای اعمال این موضوع، به باگی برخورد کردیم که پیکسل های مرزی اشتباه میشدند(رنگ سفید) و تصویر مورد نظرمون ایجاد نمیشد. این باگ به نظر از این ناشی می شد که همسایه های نامعتبر به درستی هندل نشده بودند. در نسخه جدید کد، تلاش شده تا بدون استفاده از شرط های زیاد (مثل if-else) و با استفاده از تکنیک های Clamp کردن مقادیر، مشکل Warp Divergence حل شود. این رویکرد باعث می شود که تمامی تردها در یک Warp مسیر اجرای یکسانی را دنبال کنند.

برای پیکسل های مرزی به جای استفاده از شرط های اضافی برای بررسی معتبر بودن همسایه ها یا جایگزینی آن ها با مقدار 0، از میروور کردن و استفاده از clamp مقادیر در مرز استفاده شد. همچنین دسترسی ها به حافظه داخل محدوده تصویر هستند و نیازی به شاخه بندی برای بررسی معتبر بودن همسایه ها نیست. از `__syncthreads()` نیز استفاده شد

```
neighborX = max(0, min(cols - 1, neighborX));
neighborY = max(0, min(rows - 1, neighborY));
```

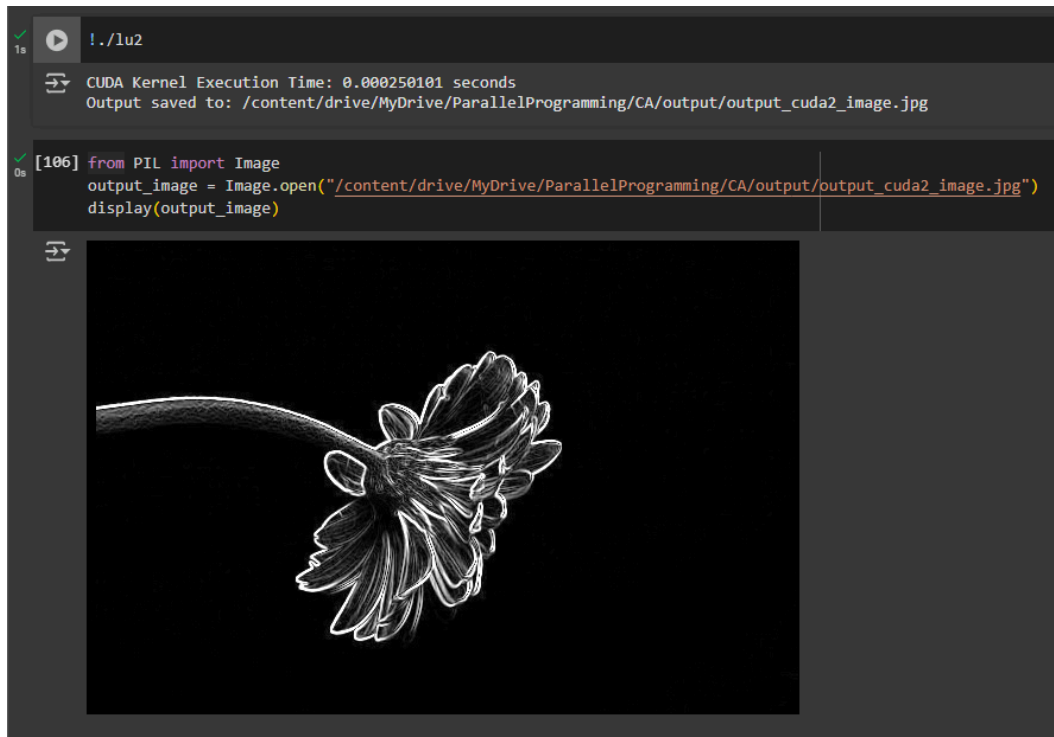
نتیجه این قسمت:



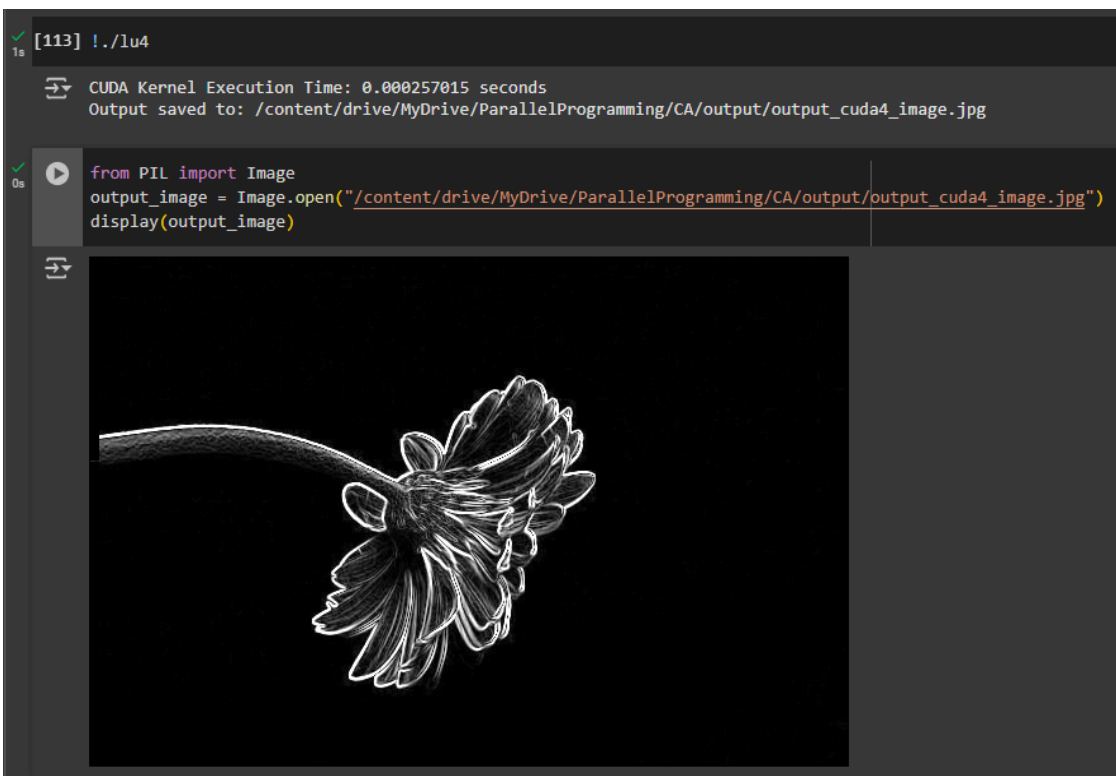
زمان گزارش شده: 0.0003349

2. اعمال loop unrolling

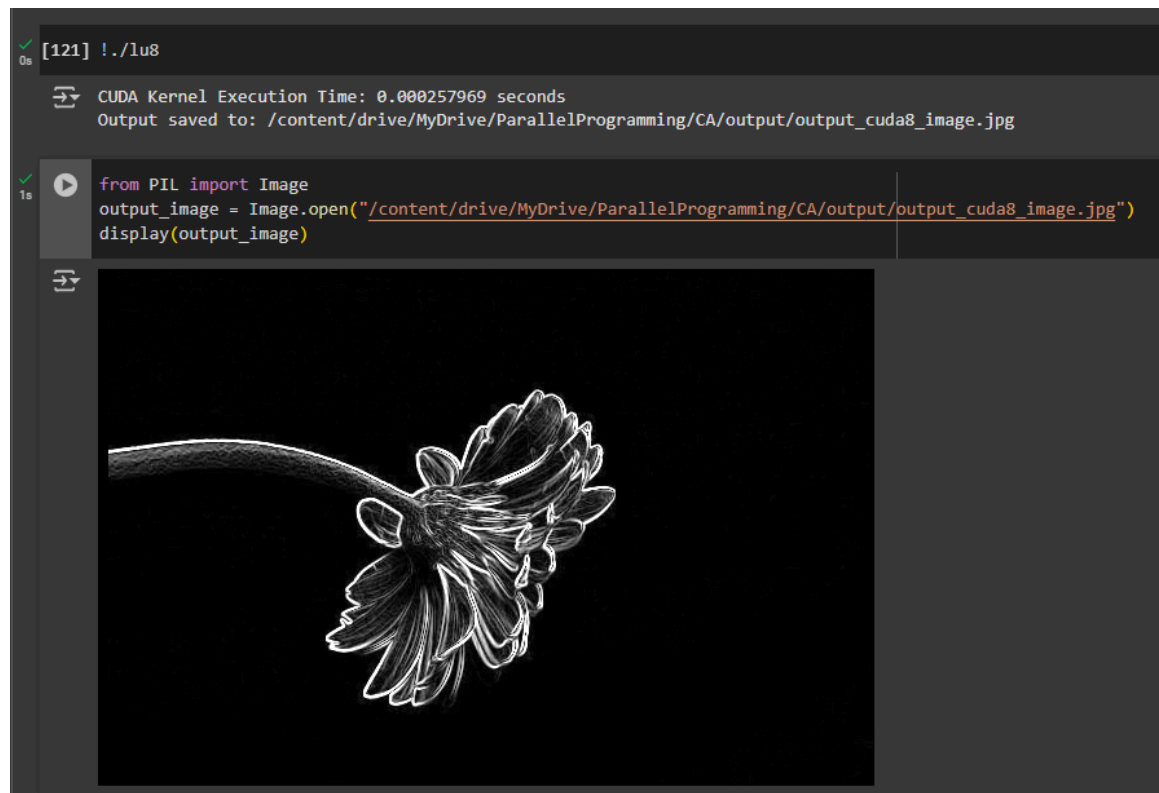
درجه 2: پردازش دو همسایه کنار هم



زمان اجرا گزارش شده: 0.0002501  
درجه 4: پردازش 4 همسایه کنار هم



زمان اجرا گزارش شده: 0.0002570  
درجه 8: پردازش 8 همسایه کنار هم



زمان گزارش شده: 0.0002579

ارزیابی عملکرد کودا:

استفاده از CUDA حتی بدون Loop Unrolling زمان اجرا را به شدت کاهش داد و نشان داد که پردازش موازی GPU برای چنین الگوریتم‌هایی بسیار کارآمد است. با اعمال Loop Unrolling، زمان اجرا بیشتر بهبود یافت. Base-2 به دلیل تعادل مناسب بین کاهش سربار حلقه‌ها و استفاده از منابع GPU، بهترین عملکرد را ارائه داد.

با افزایش درجه Loop Unrolling، مصرف رجیسترها و فشار روی حافظه GPU افزایش یافت.

Base-4 و Base-8 برای تصاویر کوچک و متوسط بهبود عملکرد قابل توجهی نسبت به Base-2 نداشتند، زیرا فشار بیشتر روی منابع GPU باعث کاهش بهره‌وری شد.

\* هر کد چندین بار اجرا شد تا ببینیم محدوده زمانی همان حدود باشد.

### توضیح کد داده شده:

کد فراهم شده در پروژه یک برنامه‌ی rendering ساده با استفاده از CUDA است که از Ray Tracing برای ایجاد تصویر استفاده می‌کند.

1. ساختار Vec3: این ساختار برای نمایش بردارهای سه‌بعدی (مانند موقعیت، جهت، رنگ و غیره) استفاده می‌شود. عملیات‌های ریاضی مانند جمع، ضرب، ضرب داخلی و نرمال‌سازی در این struct تعریف شده‌اند.

2. ساختار Ray: این ساختار یک پرتو را نشان می‌دهد که شامل یک نقطه‌ی شروع (origin) و یک جهت (direction) است. تابع at موقعیت نقطه‌ای روی پرتو را در فاصله‌ی t محاسبه می‌کند.

3. ساختار Hitable: این ساختار برای نمایش اشیاء قابل برخورد (مانند کره‌ها و صفحه) استفاده می‌شود. هر شیء شامل نوع (type)، مرکز (center)، شعاع (radius برای کره)، (normal برای صفحه) و رنگ (color) است.

4. توابع hitSphere و hitPlane: این توابع بررسی می‌کنند که آیا یک پرتو با یک کره یا صفحه برخورد می‌کند یا نه. اگر برخورد اتفاق بیفتد، فاصله‌ی برخورد (t) و نرمال سطح (normal) محاسبه می‌شوند.

5. تابع rayColor: این تابع رنگ هر پیکسل را محاسبه می‌کند. ابتدا نزدیک‌ترین شیء که با پرتو برخورد می‌کند را پیدا می‌کند، سپس نور محیطی و منتشر شده را محاسبه می‌کند. اگر نقطه در سایه باشد، نور منتشر شده صفر می‌شود.

6. تابع renderKernel: این تابع هسته‌ی CUDA است که برای هر پیکسل تصویر، تابع rayColor را فراخوانی می‌کند و رنگ پیکسل را محاسبه می‌کند.

7. تابع saveToPPM: این تابع تصویر رندر شده را در قالب فایل PPM ذخیره می‌کند.

8. تابع main: در این تابع، اشیاء صحنه (سه کره و یک صفحه) تعریف می‌شوند، سپس renderKernel فراخوانی می‌شود و در نهایت تصویر در فایل output.ppm ذخیره می‌شود.

### تغییرات ایجاد شده:

1. اضافه کردن تابع isShadowed

این تابع بررسی می‌کند که آیا نقطه‌ای که روی یک شیء (کره یا صفحه) قرار دارد، توسط نور سایه‌اندازی می‌شود یا نه. اگر پرتوی سایه (از نقطه به سمت منبع نور) با هر شیء دیگری در صحنه برخورد کند، نقطه در سایه قرار می‌گیرد.

در این کد shadow\_ray پرتو ای است از نقطه‌ی برخورد به سمت منبع نور ایجاد می‌شود. t\_max و t\_min محدوده‌ی معتبر برای برخورد پرتو با اشیاء را تعیین می‌کنند.

```
__device__ bool isShadowed(const Vec3& point, const Vec3& light_pos, Hittable* objects, int num_objects) {
    Vec3 light_dir = (light_pos - point).normalize();
    Ray shadow_ray(point + light_dir * 0.001f, light_dir); // Offset to avoid self-intersection

    float t_min = 0.001f;
    float t_max = (light_pos - point).length();

    for (int i = 0; i < num_objects; ++i) {
        float t;
        Vec3 temp_normal;
        bool hit = false;

        if (objects[i].type == SPHERE) {
            hit = hitSphere(objects[i], shadow_ray, t_min, t_max, t, temp_normal);
        } else if (objects[i].type == PLANE) {
            hit = hitPlane(objects[i], shadow_ray, t_min, t_max, t, temp_normal);
        }

        if (hit) {
            return true; // Shadowed
        }
    }
    return false; // Not shadowed
}
```

2. تغییرات در تابع rayColor

shadowed: بررسی می‌کند که آیا نقطه در سایه است یا نه.

ambient: نور محیطی است که همیشه وجود دارد (حتی در سایه).

diffuse: نور منتشر شده است که اگر نقطه در سایه باشد، صفر می‌شود.



result\_color: ترکیب نور محیطی و منتشر شده است.

```
if (hit_index >= 0) {
    Vec3 hit_point = r.at(closest_t);

    // Check if the point is in shadow
    bool shadowed = isShadowed(hit_point, light_pos, objects, num_objects);

    Vec3 light_dir = (light_pos - hit_point).normalize();
    float intensity = fmaxf(0.0f, normal.dot(light_dir));

    Vec3 ambient = 0.1f * color;
    Vec3 diffuse = shadowed ? Vec3(0, 0, 0) : intensity * color; // No diffuse if shadowed

    Vec3 result_color = ambient + diffuse;

    return result_color;
}
```

ران کردن در colab و نمایش خروجی:

```
✓ 0s !nvcc --version

nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2023 NVIDIA Corporation
Built on Tue_Aug_15_22:02:13_PDT_2023
Cuda compilation tools, release 12.2, V12.2.140
Build cuda_12.2.r12.2/compiler.33191640_0

✓ 0s %%writefile ray_tracing_shadows.cu
```

```
!nvcc ray_tracing_shadows.cu -o ray_tracing_shadows

[5] !./ray_tracing_shadows
```



```
from PIL import Image
import matplotlib.pyplot as plt

image = Image.open('output.ppm')

plt.imshow(image)
plt.axis('off')
plt.show()
```

