پروژه پنجم برنامه نویسی موازی

کسرا نوربخش ۸۱۰۱۰۰۲۳۰

شهنام فیضیان 810100197

مقدمه

در این پروژه در بخش اول تلاش کردیم تا الگوریتم Edge Detection را که پیش تر در درس هم دیده بودیم، هم به صورت سریال و هم به صورت موازی با استفاده از python multiprocessing پیاده سازی بکنیم. در بخش دوم هم به سراغ همین کار رفتیم اما این بار به کمک CUDA که تسریع بسیار جالبی را مشاهده کردیم. در بخش سوم هم کدی در اختیارمان قرار گرفته بود که آن را باید به نحوی کامل می کردیم که سایه گذاری را برای اشیا موجود در آن انجام دهد. در بخش امتیازی هم با اضافه کردن دوربینی به کد بخش سوم، زوایا دید مختلف را ساختیم. (این پروژه را با استفاده از Google Colab انجام دادیم که فایل آن PP-CA5-810100230-810100197.ipynb است و خروجی های هر بخش در فولدر output files قابل دیدن می باشد)

Python Multiprocessing

داخل صورت پروژه توضیح خوبی داده شده بود که Edge Detection با استفاده از کرنل sobel چگونه انجام می شود. ابتدا در بخش Serial Edition پیاده سازی تماماً سریال این الگوریتم را انجام دادیم با استفاده از کرنل های منحصر به فرد:

```
sobel_x = np.array([[-1, 0, 1],
                    [-2, 0, 2],
                    [-1, 0, 1]], dtype=np.float32)

sobel_y = np.array([[-1, -2, -1],
                    [ 0,  0,  0],
                    [ 1,  2,  1]], dtype=np.float32)
```

برای آن که این کرنل 3 در 3 می باشد و ممکن است که تصویر ما ابعادش بر 3 بخش پذیر نباشد، با استفاده از padding این مساله را رفع کردیم:

```
padded_image = np.pad(image, pad_width=1, mode='constant', constant_values=0)
```

در نهایت هم با استفاده از حلقه تودرتو، الگوریتم را اعمال کردیم:

```
for i in range(1, padded_image.shape[0] - 1):
    for j in range(1, padded_image.shape[1] - 1):
        region = padded_image[i-1:i+2, j-1:j+2]
        grad_x[i-1, j-1] = np.sum(region * sobel_x)
        grad_y[i-1, j-1] = np.sum(region * sobel_y)
```

نتیجه این مرحله را می توان در فایل: serial_sobel_output.jpg، مشاهده کرد. حال به سراغ مشاهده خروجی این بخش می پردازیم:

عکس 1: نمونه ورودی

عکس 2: نمونه خروجی

```
Runtime for serial Sobel filter: 9.9004 seconds
Serial Sobel output saved to serial_sobel_output.jpg
```

عکس 3: خروجی ترمینال این بخش

در هنگام اجرای برنامه سـریال، runtime های مختلفی را می گرفتیم که به عنوان نماینده، عدد 4.5567 seconds را گرفتیم. ضـمناً همان طور که در گروه درس مطرح شـده بود، برای مقایسـه ها در طول پروژه از ورودی که داده شـد استفاده کردیم:

عکس 4: ورودی اصلی

عکس 5: خروجی عکس قبل

```
--2025-01-10 11:18:25--  https://media.geeksforgeeks.org/wp-content/uploads/20211220204344/pexelsylanitekoppens23431701.jpg
Resolving media.geeksforgeeks.org (media.geeksforgeeks.org)... 18.160.78.91, 18.160.78.26, 18.160.78.21, ...
Connecting to media.geeksforgeeks.org (media.geeksforgeeks.org)|18.160.78.91|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 90599 (88K) [image/jpeg]
Saving to: ‘flower.jpg’

flower.jpg          100%[===================>]  88.48K  --.-KB/s    in 0.02s

2025-01-10 11:18:25 (3.54 MB/s) - ‘flower.jpg’ saved [90599/90599]

Runtime for serial Sobel filter: 3.7435 seconds
Serial Sobel output saved to serial_sobel_output.jpg
```

عکس 6: خروجی ترمینال این بخش

حال به سراغ پیاده سازی با چند پردازه برویم. کد این قسمت در بخش Parallel Edition موجود است. در این قسمت ما تابع process_chunk را داریم که در واقع یک chunk یا همان بخشی از عکس را می گیرد و کاملاً شبیه با نسخه سریال که پیش تر داشتیم، پردازش را انجام می دهد. با استفاده از تابع process_image_with_multiprocessing که 2 ورودی عکس و تعداد پردازه ها را می گیرد، عکس را چند بخش که هر کدام شامل چند سطر هستند، تقسیم می کنیم و به پردازه ها اختصاص می دهیم. فرضاً اگر ما 4 پردازه داریم، ¼ بالایی عکس به یک پردازه، ¼ پایینی به یک پردازه و به همین ترتیب تقسیم کار انجام می شود. به خاطر آن که خروجی ما صحیح نبود و خط سفیدی میان مرز های هر پردازه ایجاد می شد، ما به این صورت عمل کردیم که پنجره ها

با chunk قبلی و بعدی یک overlap ای داشته باشند. با استفاده از این حلقه هم توانستیم به درستی نتیجه های نهایی را ادغام کنیم:

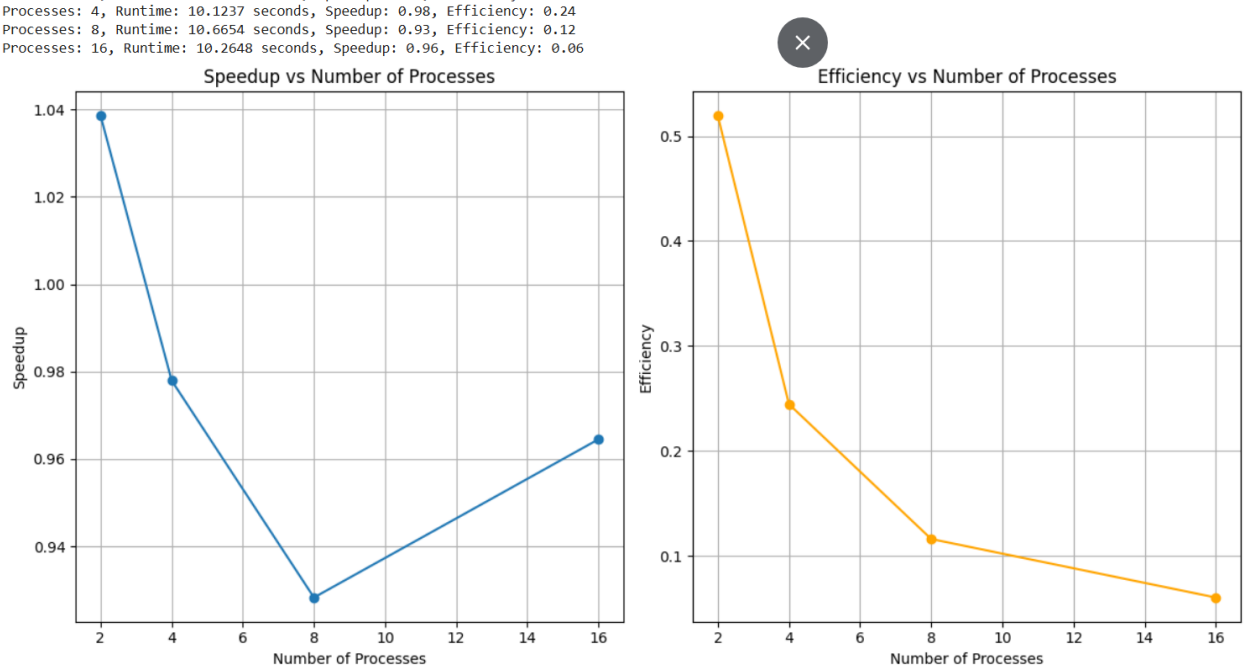
```
for i, (grad_x_chunk, grad_y_chunk) in enumerate(results):
    if i == 0:
        grad_x_chunks.append(grad_x_chunk[:-1])
        grad_y_chunks.append(grad_y_chunk[:-1])
    elif i == num_processes - 1:
        grad_x_chunks.append(grad_x_chunk[1:])
        grad_y_chunks.append(grad_y_chunk[1:])
    else:
        grad_x_chunks.append(grad_x_chunk[1:-1])
        grad_y_chunks.append(grad_y_chunk[1:-1])
```

این بخش را با 2، 4، 8 و 16 پردازه اجرا کردیم و به نتایج متفاوتی رسیدیم. و در آخر هم نمودار های خواسته شده را رسم کردیم تا ببینیم به چه میزان تسریع و بهره وری داریم که مشاهده شد که زیاد کردن تعداد پردازه ها به این خاطر که زمان بر هستند، ممکن است اندکی به ما تسریع بدهند اما به لحاظ بهره وری اصلاً جالب نیست.

نتیجه های این مرحله را می توان در فایل های: output_"numProcess"_processes.jpg، مشاهده کرد. حال به سراغ مشاهده خروجی این بخش می پردازیم (به ترتیب از چپ به راست با استفاده از 2، 4، 8 و 16 پردازه):

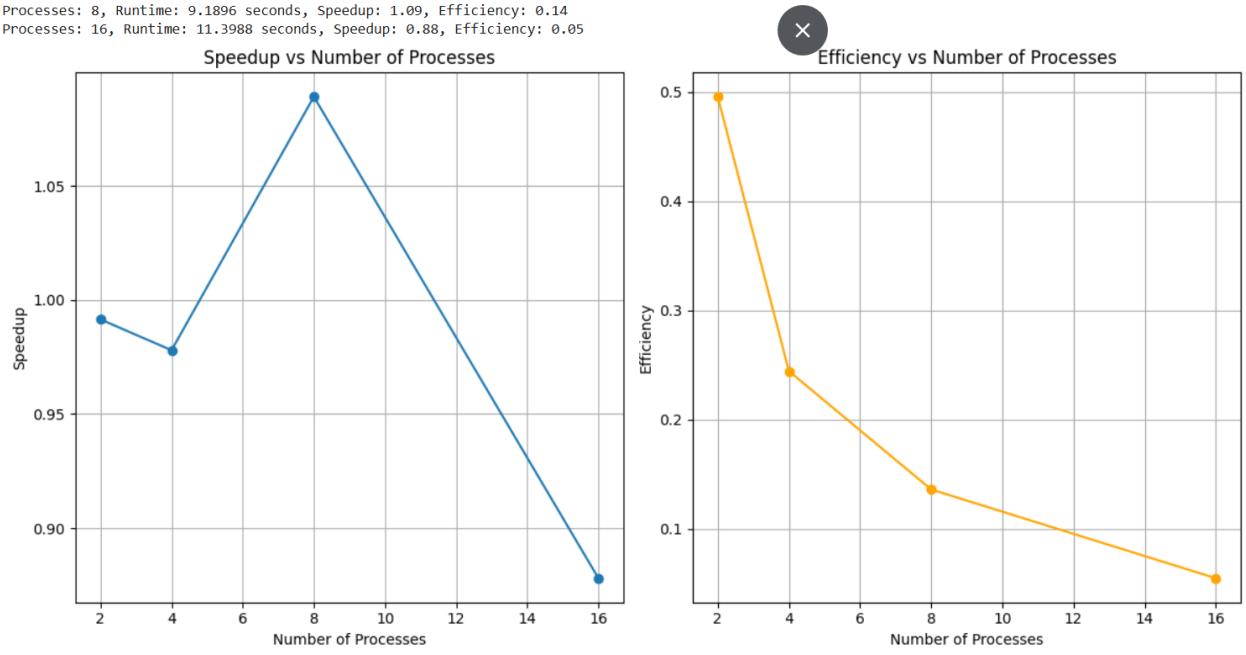
با مشاهده خروجی های صفحه بعدی میفهمم که برای عکس من بهترین تسریع را با 8 پردازه داریم اما بهره وری ها اصلاً جالب نیست و به نوعی موازی نکردن با عکس من بهترین راه حل است.

```
Processes: 2, Runtime: 9.5327 seconds, Speedup: 1.04, Efficiency: 0.52
Processes: 4, Runtime: 10.1237 seconds, Speedup: 0.98, Efficiency: 0.24
Processes: 8, Runtime: 10.6654 seconds, Speedup: 0.93, Efficiency: 0.12
Processes: 16, Runtime: 10.2648 seconds, Speedup: 0.96, Efficiency: 0.06
```

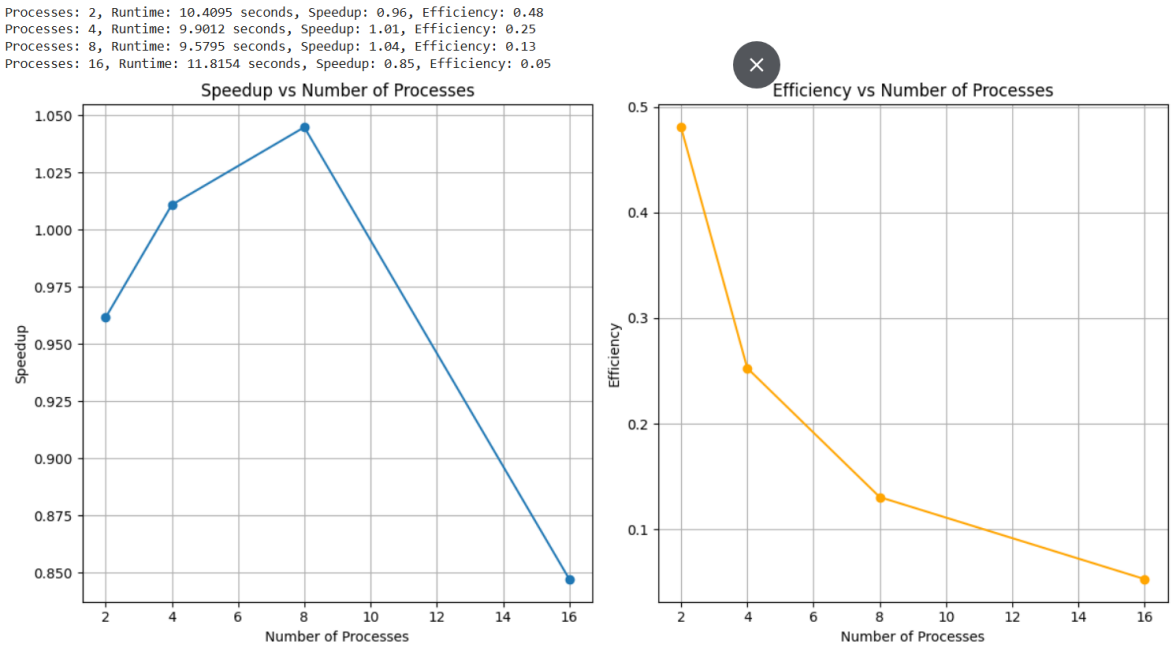


عکس 7: نمونه ای از خروجی این بخش با استفاده از عکس خودم

```
Processes: 2, Runtime: 10.0944 seconds, Speedup: 0.99, Efficiency: 0.50
Processes: 4, Runtime: 10.2339 seconds, Speedup: 0.98, Efficiency: 0.24
Processes: 8, Runtime: 9.1896 seconds, Speedup: 1.09, Efficiency: 0.14
Processes: 16, Runtime: 11.3988 seconds, Speedup: 0.88, Efficiency: 0.05
```



عکس 8: نمونه دیگر خروجی

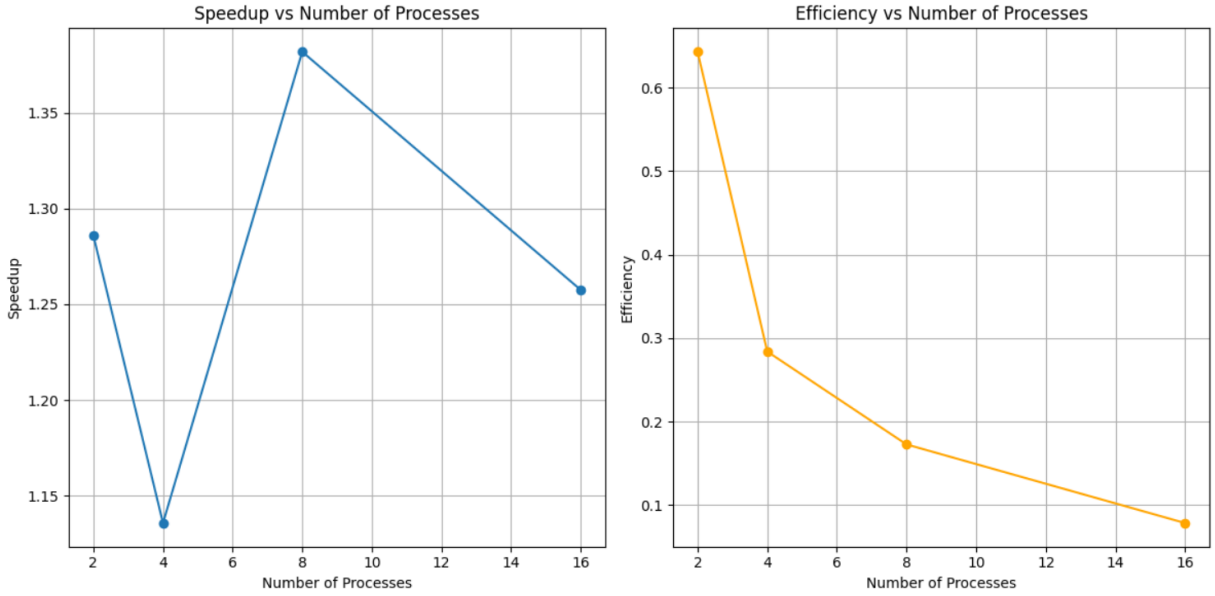


عکس 9: نمونه خروجی آخر

حال به سراغ مشاهده خروجی ها با عکس اصلی می پردازیم (به ترتیب از چپ به راست با استفاده از 2، 4، 8 و 16 پردازه):

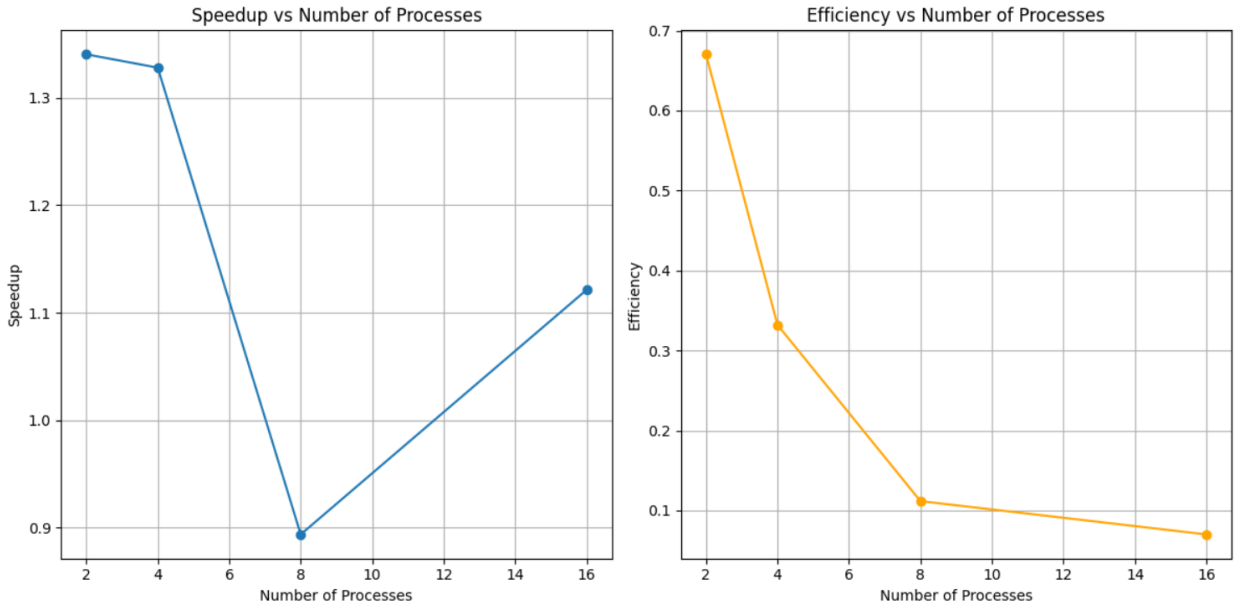
با مشاهده خروجی های صفحه بعد میفهمم که برای عکس اصلی بهترین تسریع را با تعداد مختلفی پردازه به دست می آوریم اما بهره وری مان فقط با 2 پردازه قابل توجیه است.

```
Processes: 2, Runtime: 3.7990 seconds, Speedup: 1.29, Efficiency: 0.64
Processes: 4, Runtime: 4.3007 seconds, Speedup: 1.14, Efficiency: 0.28
Processes: 8, Runtime: 3.5351 seconds, Speedup: 1.38, Efficiency: 0.17
Processes: 16, Runtime: 3.8846 seconds, Speedup: 1.26, Efficiency: 0.08
```



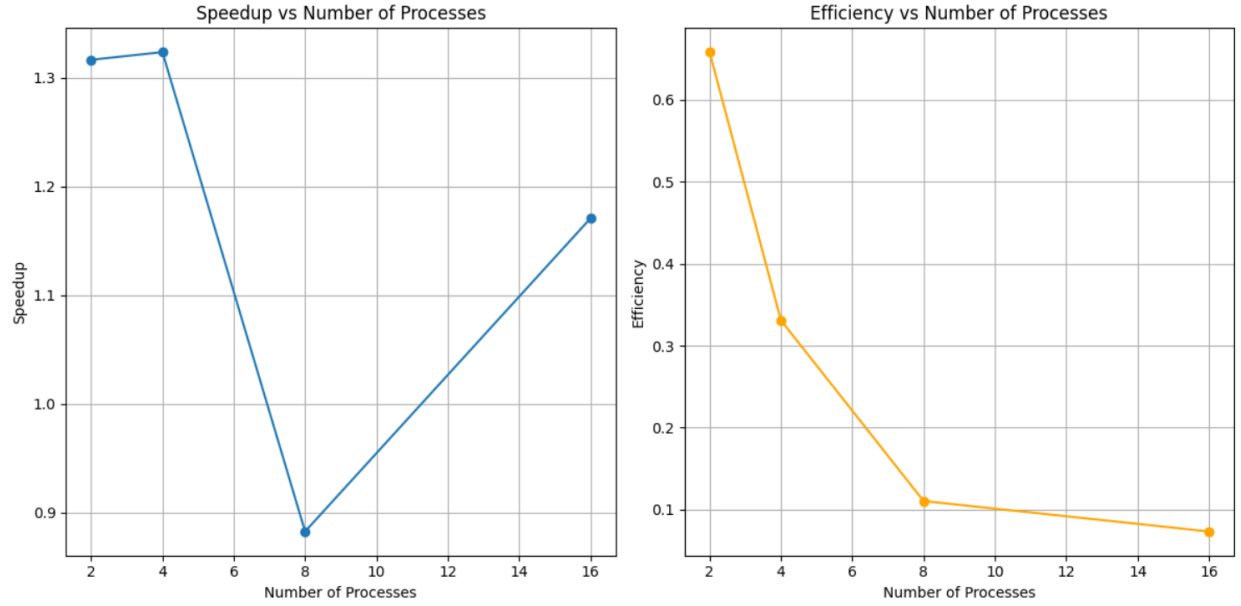
عکس 10: نمونه ای خروجی با استفاده از عکس اصلی

```
Processes: 2, Runtime: 3.3991 seconds, Speedup: 1.34, Efficiency: 0.67
Processes: 4, Runtime: 3.4309 seconds, Speedup: 1.33, Efficiency: 0.33
Processes: 8, Runtime: 5.0986 seconds, Speedup: 0.89, Efficiency: 0.11
Processes: 16, Runtime: 4.0636 seconds, Speedup: 1.12, Efficiency: 0.07
```



عکس 11: نمونه دیگر خروجی

```
Processes: 2, Runtime: 3.4616 seconds, Speedup: 1.32, Efficiency: 0.66
Processes: 4, Runtime: 3.4431 seconds, Speedup: 1.32, Efficiency: 0.33
Processes: 8, Runtime: 5.1613 seconds, Speedup: 0.88, Efficiency: 0.11
Processes: 16, Runtime: 3.8925 seconds, Speedup: 1.17, Efficiency: 0.07
```



عکس 12: نمونه خروجی آخر

CUDA

حال در این بخش به سراغ انجام دادن همان کار بخش قبلی به کمک CUDA رفتیم که تسریع شگفت انگیزی را دیدیم. ابتدا پیاده سازی های متعددی انجام دادیم اما در نهایت به نسخه آخر رسیدیم که هم مساله Divergence را ندارد و هم اینکه حلقه اعمال کردن sobel را unroll کردیم. ضمناً این پیاده سازی نهایی با block-size های مختلف هم اجرا شد. دقت شد که نحوه ای که در اینجا عکس پردازش می شود، یعنی به صورت سطری، کاملاً منتطبق با بخش قبل باشد. در اینجا ما تعدادی block داریم (16) که هر کدام متناسب با بخش های عکس، thread دارند و محاسبات را انجام می دهند، در واقع هر Thread یک سطر مربوط به خودش را پردازش می کند. کرنل ما sobel_filter_rowwise_unrolled می باشد که مشابه همان حلقه قبلی را دارد و محاسبات را انجام می دهد:

```
for (int col = 0; col < width; ++col)
    float grad_x = 0.0f;
    float grad_y = 0.0f;
    int idx = 0;
    for (int i = -1; i <= 1; ++i) {
        for (int j = -1; j <= 1; ++j, ++idx) {
            int r = max(0, min(height - 1, row + i));
            int c = max(0, min(width - 1, col + j));
            float pixel_value = static_cast<float>(input[r * width + c]);
            grad_x += pixel_value * sobel_x[idx];
            grad_y += pixel_value * sobel_y[idx];
```

با اســـتفاده از cudaEvent زمان اجرای این بخش را محاســـبه کردیم و نتیجه این مرحله را می توان در فایل: output_cuda_unrolled.jpg، مشاهده کرد. حال به سراغ دیدن خروجی ها میرویم:

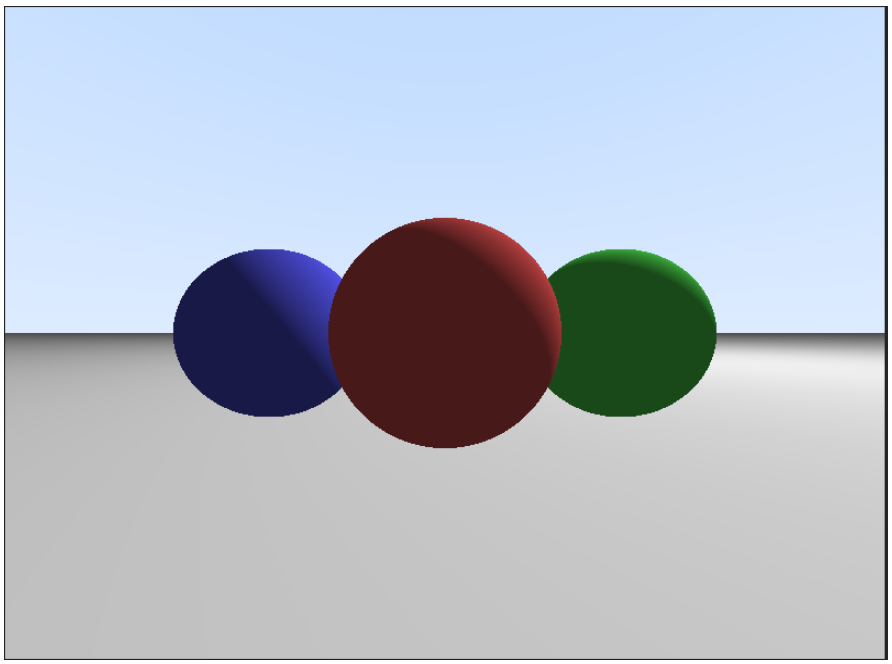
عکس 13: خروجی با CUDA

زمان اجرا های متغییری را دیدیم:

```
CUDA Sobel filter applied in 130.682 ms.
CUDA Sobel filter applied in 147.54 ms.
CUDA Sobel filter applied in 0.647616 ms.
CUDA Sobel filter applied in 0.58144 ms.
```

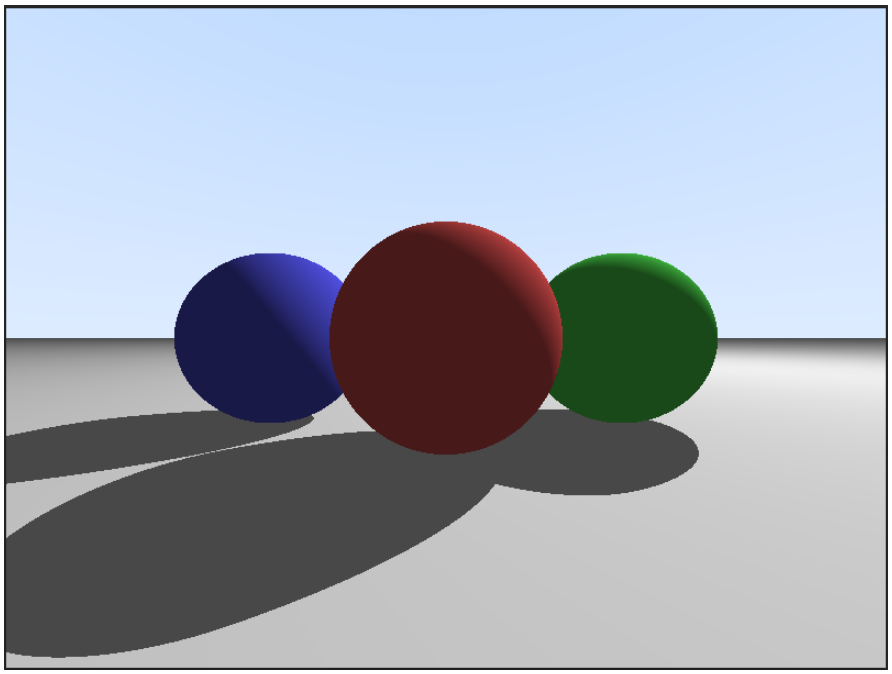
Graphical

در این بخش ابتدا با اجرا کردن کد گوی ها رندر می شدند:



عکس 14: خروجی اولیه

که سپس با سایه گذاری به این نتیجه رسیدیم:



عکس 15: خروجی نهایی

کدی که در این بخش کامل شد به صورت زیر می باشد که متناسب با منبع نور، سایه ایجاد می کند و همچنین بررسی می شود که آیا شی ای جلوی نور را گرفته است یا نه (حالتی که سایه ها در هم می روند):

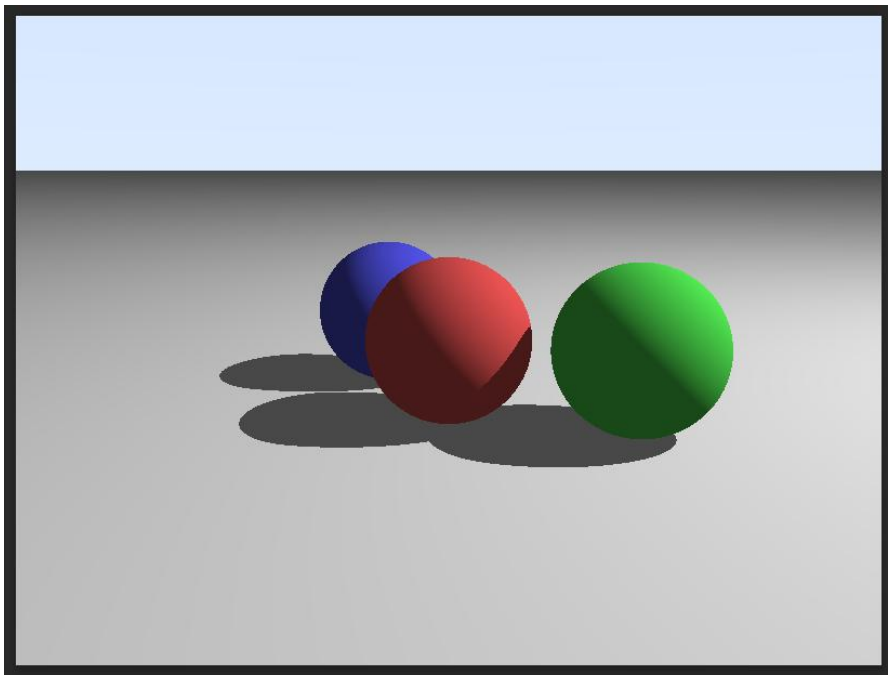
```
if (hit_index >= 0) {
    // Shadow Casting: Create a ray from the hit point to the light source
    Vec3 hit_point = r.at(closest_t);
    Vec3 light_dir = (light_pos - hit_point).normalize();
    Ray shadow_ray(hit_point, light_dir);

    // Check if any object is blocking the light
    bool is_shadowed = false;
    for (int i = 0; i < num_objects; ++i) {
        if (objects[i].type == SPHERE) {
            float t;
            Vec3 temp_normal;
            if (hitSphere(objects[i], shadow_ray, t_min, 1.0f, t, temp_normal)) {
                is_shadowed = true;
                break;
            }
        } else if (objects[i].type == PLANE) {
            float t;
            Vec3 temp_normal;
            if (hitPlane(objects[i], shadow_ray, t_min, 1.0f, t, temp_normal)) {
                is_shadowed = true;
                break;
```

```
            }
        }
    }

    // If not shadowed, compute lighting intensity
    if (!is_shadowed) {
        float intensity = fmaxf(0.0f, normal.dot(light_dir));
        Vec3 ambient = 0.1f * color;
        Vec3 diffuse = intensity * color;

        Vec3 result_color = ambient + diffuse;
        return result_color;
    } else {
        // If shadowed, return only ambient light
        return 0.1f * color;
    }
}
```

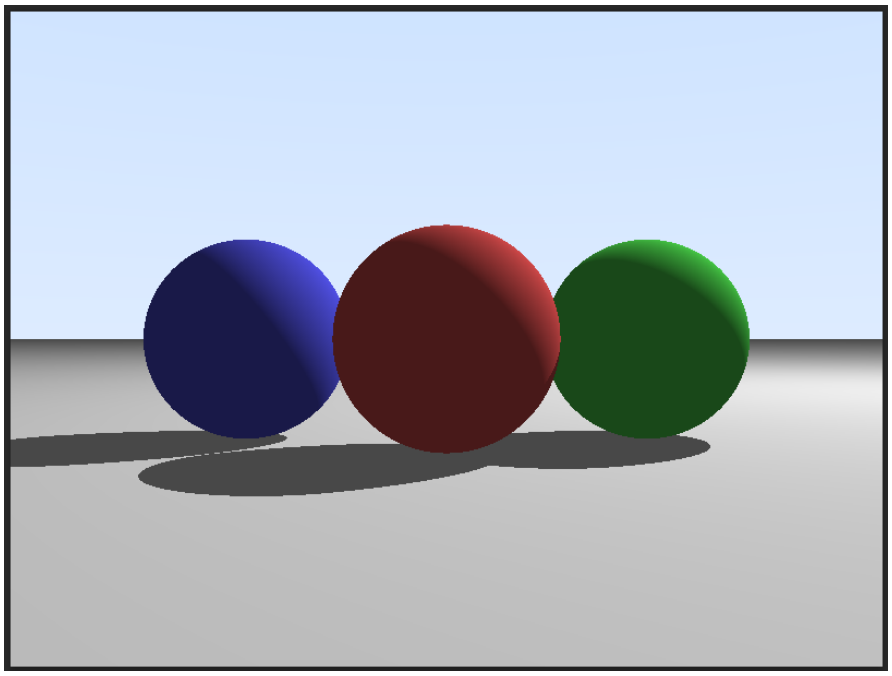
نتیجه این مرحله را می توان در فایل: output.ppm، مشاهده کرد.

Extra

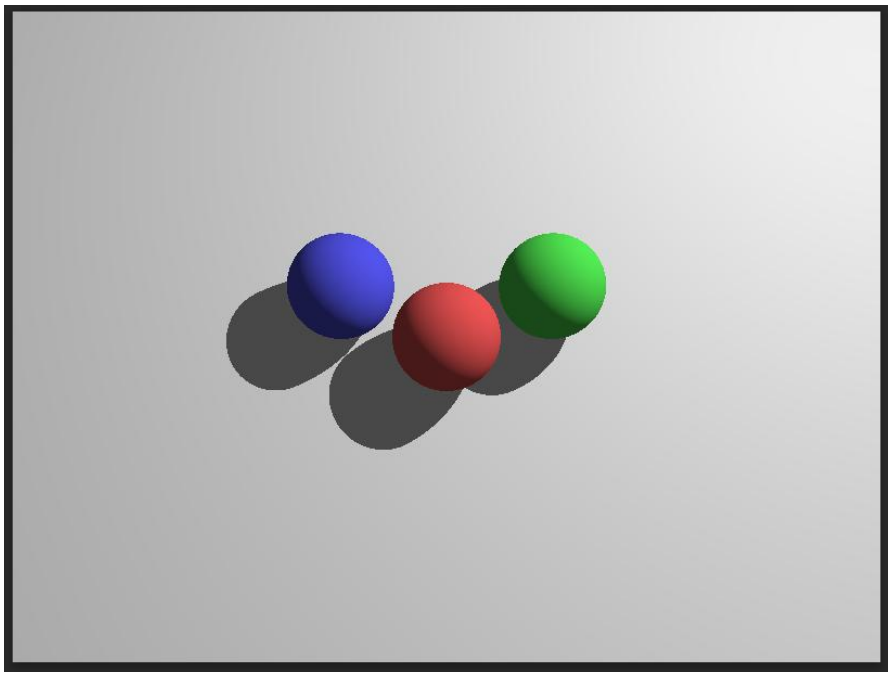
در این بخش هم تلاش شد تا دوربین گفته شده را با مشخصات داده شده به کد اضافه کنیم و در زوایا و حالت های مختلفی خروجی ها را ببینیم.



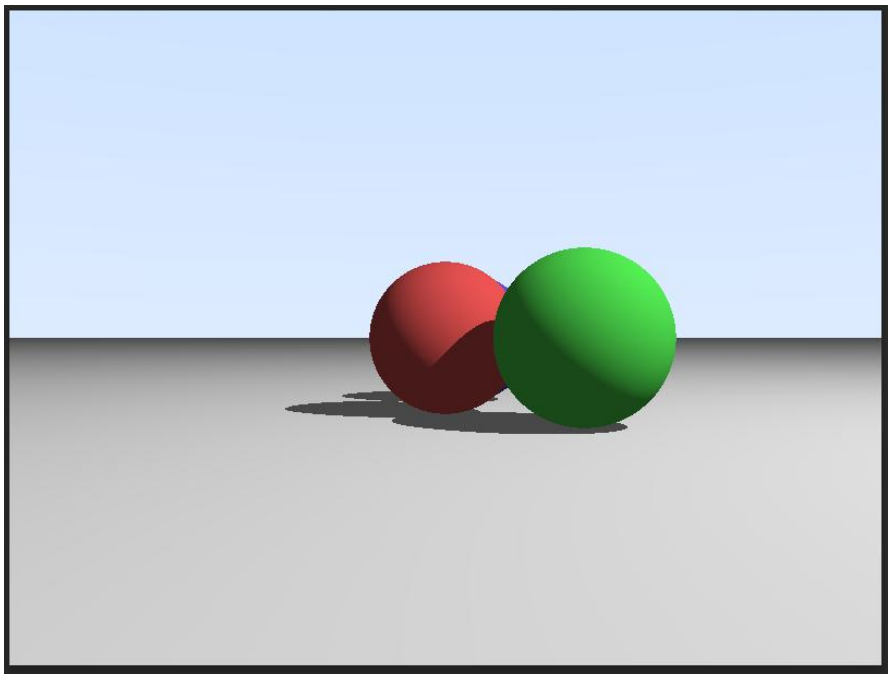
عکس 16: نمونه خروجی



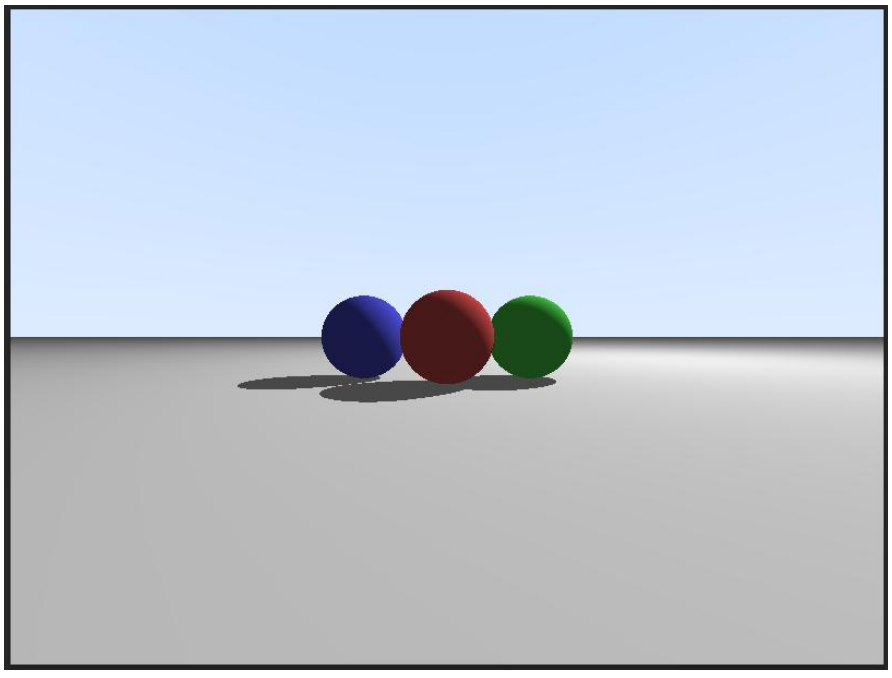
عکس 17: نمونه خروجی



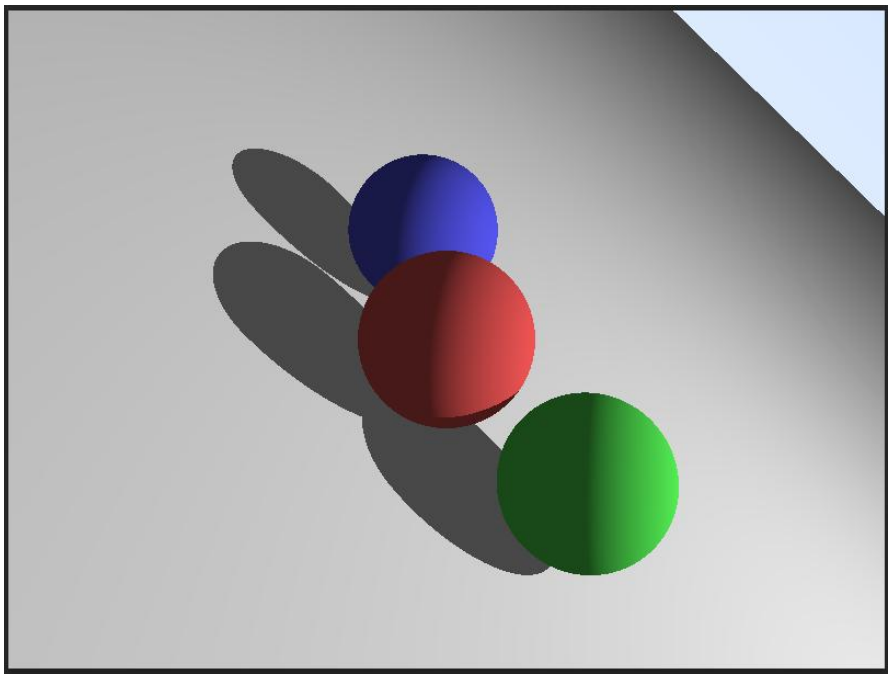
عکس 18: نمونه خروجی



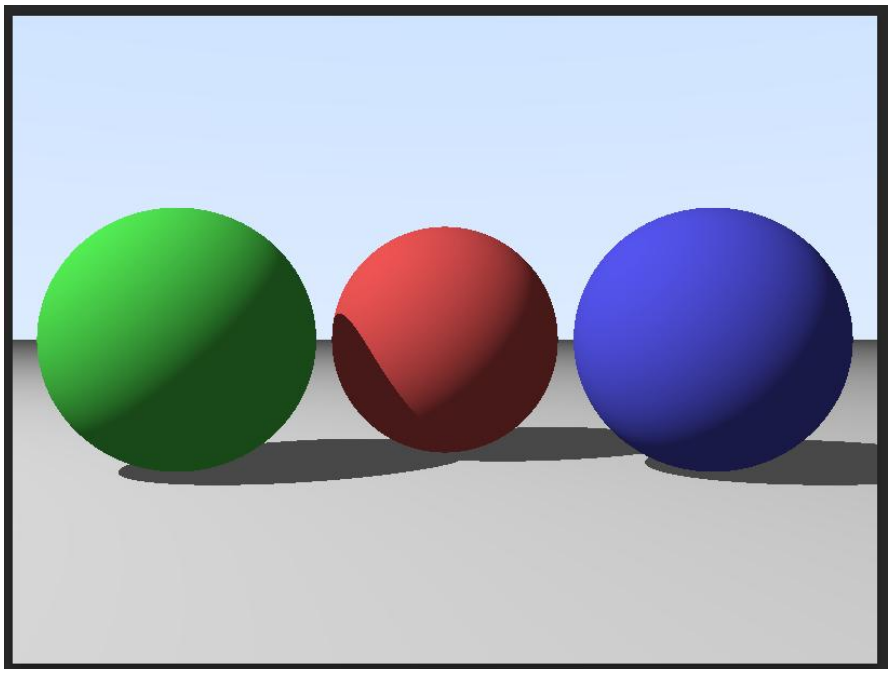
عکس 19: نمونه خروجی



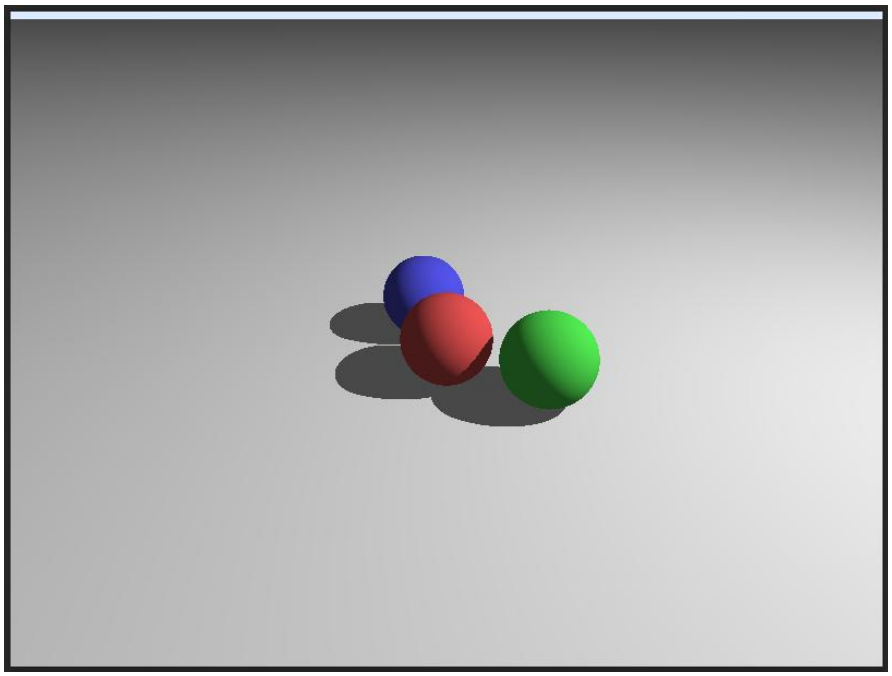
عکس 20: نمونه خروجی



عکس 21: نمونه خروجی



عکس 22: نمونه خروجی



عکس 23: نمونه خروجی

در این بخش camera struct را مطابق با جزییات گفته شده یعنی پارامتر ها و محاسبه بردار، تعریف کردیم:

```
Vec3 origin;      // Camera position (lookfrom)
Vec3 lower_left;  // Bottom-left corner of the image plane
Vec3 horizontal;  // Horizontal span of the image plane
Vec3 vertical;    // Vertical span of the image plane
__host__ __device__ Camera(const Vec3& lookfrom, const Vec3& lookat, const Vec3& up, float vfov, float aspect_ratio) {
    float theta = vfov * M_PI / 180.0f; // Convert vertical FOV to radians
    float viewport_height = 2.0f * tanf(theta / 2.0f);
    float viewport_width = aspect_ratio * viewport_height;
    Vec3 w = (lookfrom - lookat).normalize();
    Vec3 u = up.cross(w).normalize();
    Vec3 v = w.cross(u);
    origin = lookfrom;
    horizontal = viewport_width * u;
    vertical = viewport_height * v;
    lower_left = origin - horizontal / 2 - vertical / 2 - w;
}
__host__ __device__ Ray getRay(float u, float v) const {
    return Ray(origin, lower_left + u * horizontal + v * vertical - origin);
```

بخش رندر کردن هم که پیش تر به صـورت hard code بود به این صـورت عوض شـد تا با camera رندر گرفته شود (update شدن getRay هم در اینجاست):

```
__global__ void renderKernel(Vec3* pixels, int width, int height, Hittable* objects, int num_objects, Vec3 light_pos, Camera camera) {
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;
    if (x >= width || y >= height) return;
    int index = y * width + x;
    float u = (float(x) + 0.5f) / float(width);
    float v = (float(y) + 0.5f) / float(height);
    Ray r = camera.getRay(u, v);
    Vec3 color = rayColor(r, objects, num_objects, light_pos);
    color = Vec3(sqrtf(color.x), sqrtf(color.y), sqrtf(color.z)); // Gamma correction
    pixels[index] = color;
}
```

نتیجه های این مرحله را به ترتیب، می توان در فایل های: output”number”.ppm، مشاهده کرد. ضمناً برای تولید آن ها در داخل main تنظیمات مختلفی برای پارامتر های lookfrom, lookat, up, vfov در نظر گرفته شـد مثلاً برای خروجی های شماره 1 و 2 به ترتیب:

lookfrom(3.0f, 1.0f, 2.0f)

lookat(0.0f, 0.0f, -1.5f)

up(0.0f, 1.0f, 0.0f)

vfov = 45.0f

lookfrom(0.0f, 0.0f, 2.0f)

lookat(0.0f, 0.0f, -1.5f)

up(0.0f, 1.0f, 0.0f)

vfov = 45.0f

ضـمناً بخش چاپ کردن و یا ذخیره کردن خروجی های تمامی کد های تمامی بخش ها در نوت بوک کامنت شـده اند و برای اجرای برنامه خوب است که ابتدا uncomment شوند و سپس بلوک ها تک به تک run شوند.