به نام خدا

دانشگاه تهران

دانشکده مهندسی برق و کامپیوتر

برنامه نویسی موازی

گزارش کار پروژه‌ی پنجم

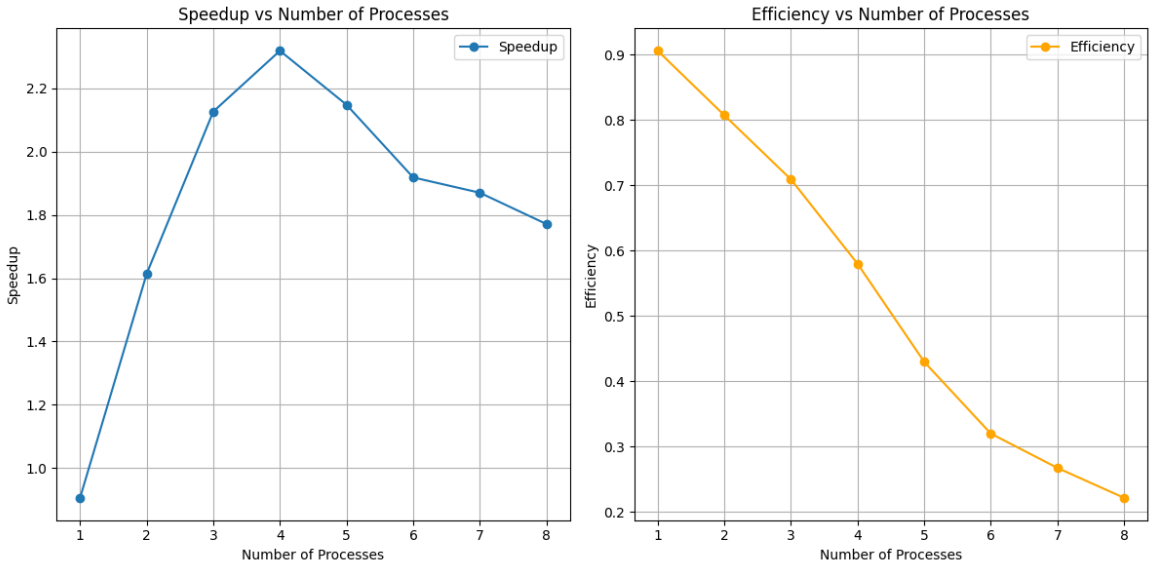
برنامه نویسی CUDA

نام و نام خانوادگی	امیرحسین ثمودی – آرمین قاسمی
شماره دانشجویی	810100198 – 810100108
تاریخ ارسال گزارش	1403/10

فهرست گزارش سوالات

بخش اول: پیاده سازی الگوریتم با python Multiprocessing

در این سوال کرنل emboss را روی تصویر اجرا می کنیم. این کرنل به تصویر افکتی سه بعدی میدهد که باعث میشود فرض کنیم این تصویر عمق دارد. در این سوال کرنل را بر روی هر سه کانال R,G,B اجرا کرده ایم و بنابراین انتظار داریم تصویر خروجی، رنگی باشد. در این کد ما تابعی داریم که با توجه به تعداد process ها و اندازه تصویر، تصویر ورودی را به بلوک های کوچکتر میشکند تا هر یک از این بلوک ها توسط یک process اجرا شوند. در روند اصلی برنامه نیز کد را یک بار به صورت سریال و چندین بار به صورت موازی با تعداد process های مختلف ران میکنیم تا نهایتا بتوانیم خروجی را plot کنیم. پس از آن با توجه به داده های جمع آوری شده خروجی را plot می کنیم.



با توجه با این plot می توانیم بگوییم efficiency با افزایش تعداد process ها همواره در حال کاهش است زیرا هیچ گاه به آن نسبت که تعداد process ها را افزایش میدهیم، speedup دریافت نمی کنیم. همینطور speedup نیز بعد از اینکه تعداد process ها از 4 بیشتر میشود، شروع به کم شدن می کند. احتمالا علت این امر این است که سربار جابه جایی داده و سیستم عامل بر speedup پردازش موازی غلبه کرده است. در پایین نمونه خروجی اجرای سریال، یکی از اجرا های موازی و تصویر اصلی آورده شده است:

شکل 1: تصویر اصلی

شکل 2: خروجی سریال

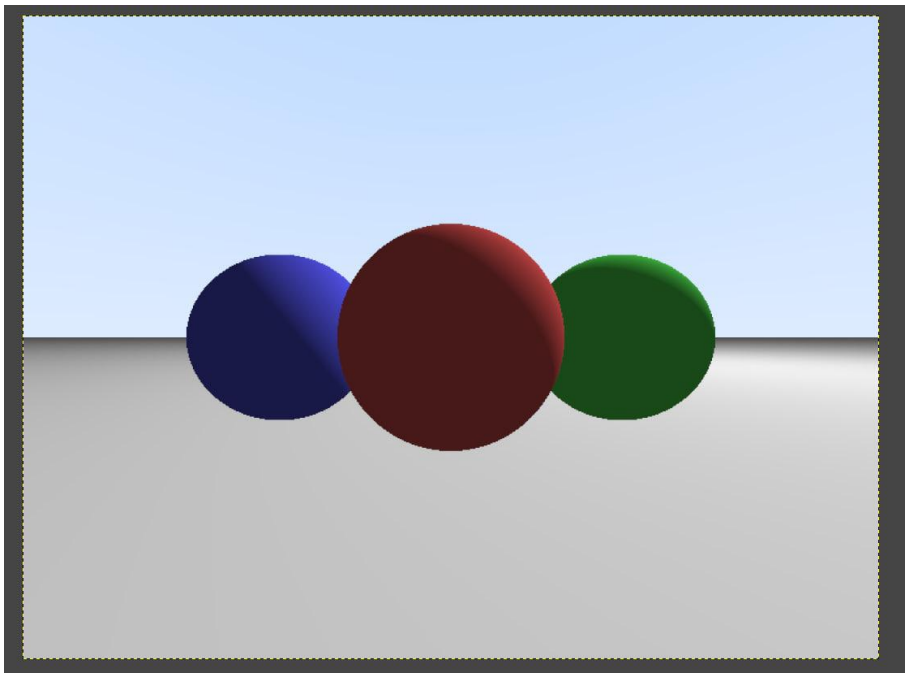
شکل 1خروجی موازی با 2 **process**

بخش دوم: پیاده سازی الگوریتم با استفاده از CUDA

ابتدا

بخش سوم: پیاده سازی گرافیکی – رندر تصویر کره از زوایای مختلف

در این بخش کد آماده داده شده را ابتدا اجرا میکنیم. این کد یک تصویر 3 بعدی را با استفاده از کتابخانه های CUDA رندر مینماید.



تصویر بالا خروجی اولیه کد را نشان میدهد. ما قصد داریم با کامل کردن تابع rayColor به تصویر سایه اضافه نماییم. کد کامل شده این تابع مطابق شکل زیر میباشد. در ادامه هر یک از بخش های آن را توضیح میدهیم.

```
__device__ Vec3 rayColor(const Ray& r, Hittable* objects, int num_objects, Vec3 light_pos) {
    float t_min = 0.001f;
    float t_max = 1e20f;
    float closest_t = t_max;
    Vec3 color(0, 0, 0);
    Vec3 normal;
    int hit_index = -1;

    // Find the closest hit
    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], r, t_min, closest_t, t, temp_normal);
        } else if (objects[i].type == PLANE) {
            hit = hitPlane(objects[i], r, t_min, closest_t, t, temp_normal);
        }

        if (hit) {
            closest_t = t;
            normal = temp_normal;
            color = objects[i].color;
            hit_index = i;
        }
    }

    if (hit_index >= 0) {
        Vec3 hit_point = r.at(closest_t);
        Vec3 light_dir = (light_pos - hit_point).normalize();

        // Create shadow ray
        Ray shadow_ray(hit_point + normal * 0.001f, light_dir);
        bool in_shadow = false;

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

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

            if (in_shadow) {
                break;
            }
        }

        float intensity = 0.0f;
        if (!in_shadow) {
            intensity = fmaxf(0.0f, normal.dot(light_dir));
        }

        Vec3 ambient = 0.1f * color;
        Vec3 diffuse = intensity * color;

        return ambient + diffuse;
    }

    // Background color
    Vec3 unit_direction = r.direction.normalize();
    float t = 0.5f * (unit_direction.y + 1.0f);
    return (1.0f - t) * Vec3(1.0f, 1.0f, 1.0f) + t * Vec3(0.5f, 0.7f, 1.0f); // Sky gradient
}
```

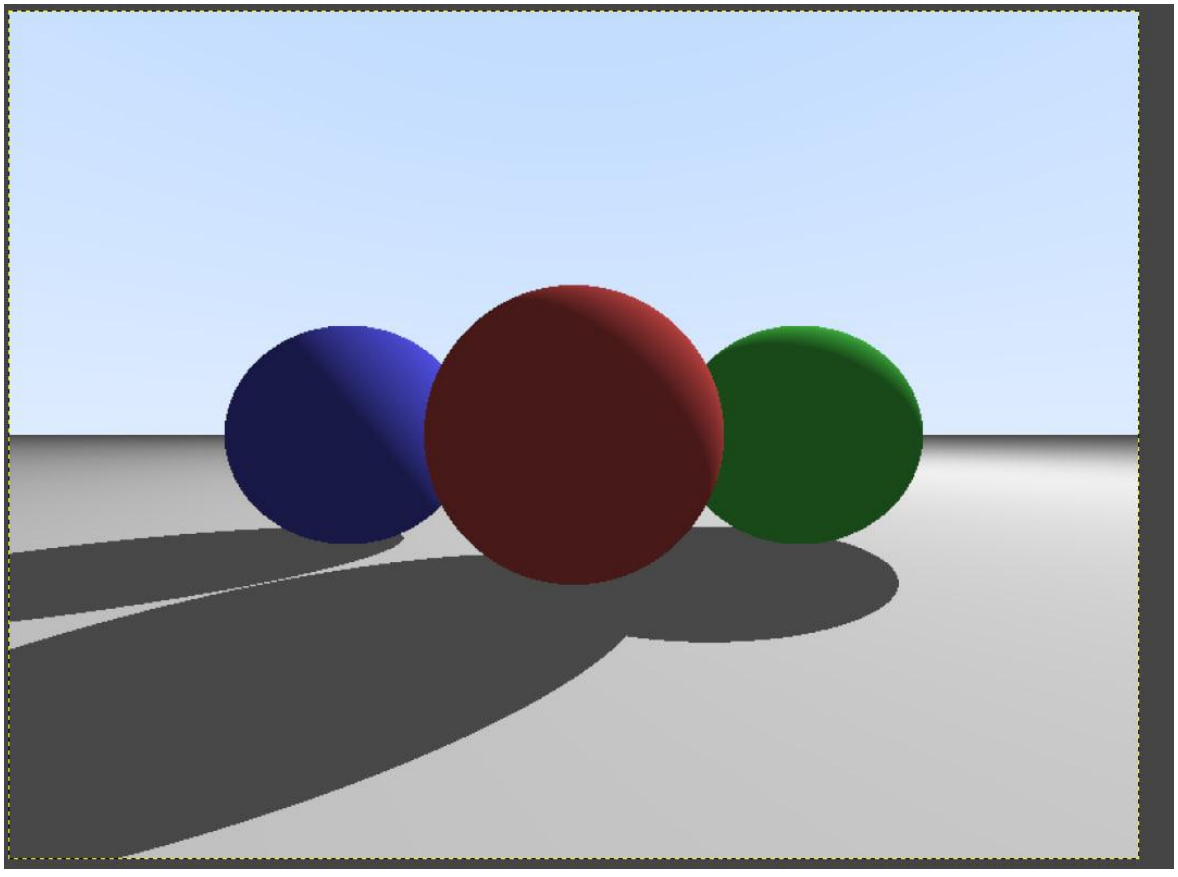
ابتدا در حلقه اول روی شی ها بررسی کرده و نزدیک ترین شی ای که با پرتو برخورد دارد را شناسایی میکنیم (با تابع hitSphere یا hitPlanel) .

حال در صورتی که برخوردی با یک شی داشت Hit point و Light direction را برای آن محاسبه میکنیم. سپس یک پرتو جدید ایجاد کرده که از hit point شروع شده و در جهت منبع نور است.

سپس برای پرتو جدید بررسی میکنیم که به شی ای برخورد دارد یا نه به همان صورت قبل و در صورت برخورد میفهمیم که منبع نور بلاک شده است و نیازی به ادامه محاسبات این پرتو نیست.

در صورتی که نقطه در سایه نیست باید intensity نور را محاسبه نماییم (ضرب داخلی surface normal و light_dir)

سپس برای تعیین رنگ ambient و diffused component را ترکیب مینماییم. خروجی نهایی به صورت زیر میباشد.



مشاهده میکنیم که سایه های کره ها به زیبایی ایجاد شده اند!