In the name of God

Project 5 — Parallel Programming Course

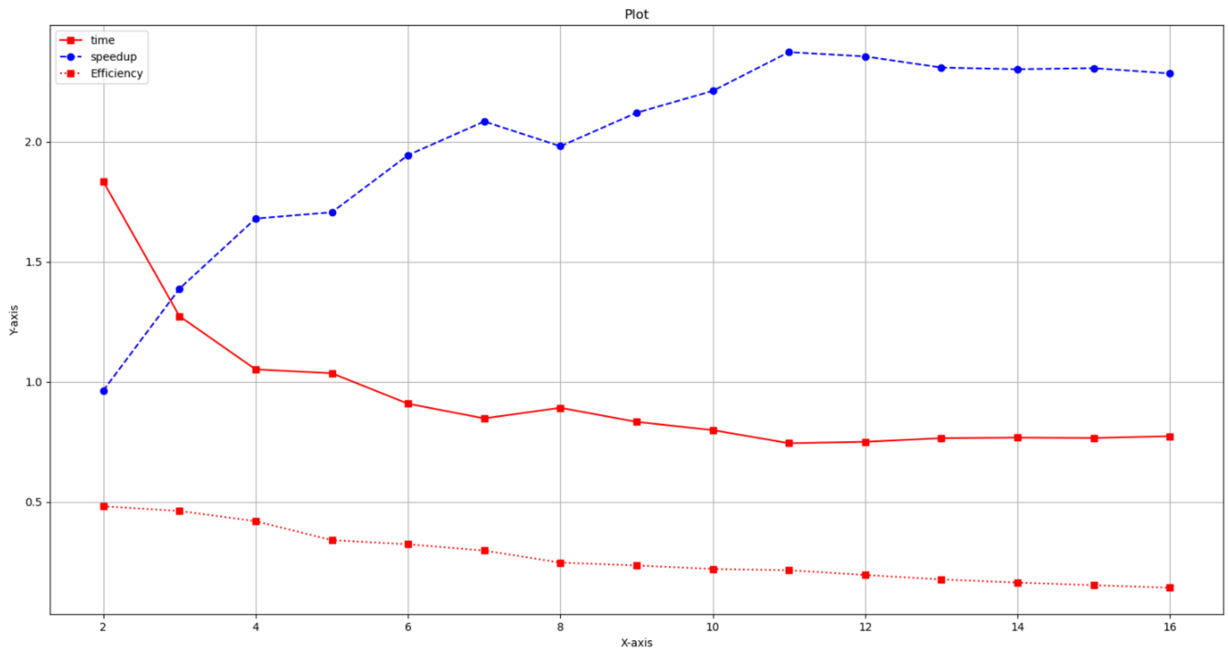
Instructor: **Dr. Safari**

Students:

Mohamad Yahyapour

Mohammad Moien Joneidi Jaafari

Part 1



In general, as the number of **processing threads** increases, the **speedup** rises progressively until it reaches a **saturation point**. Beyond this point, increasing the number of threads only slightly reduces the computation time, causing the **efficiency** to drop significantly.

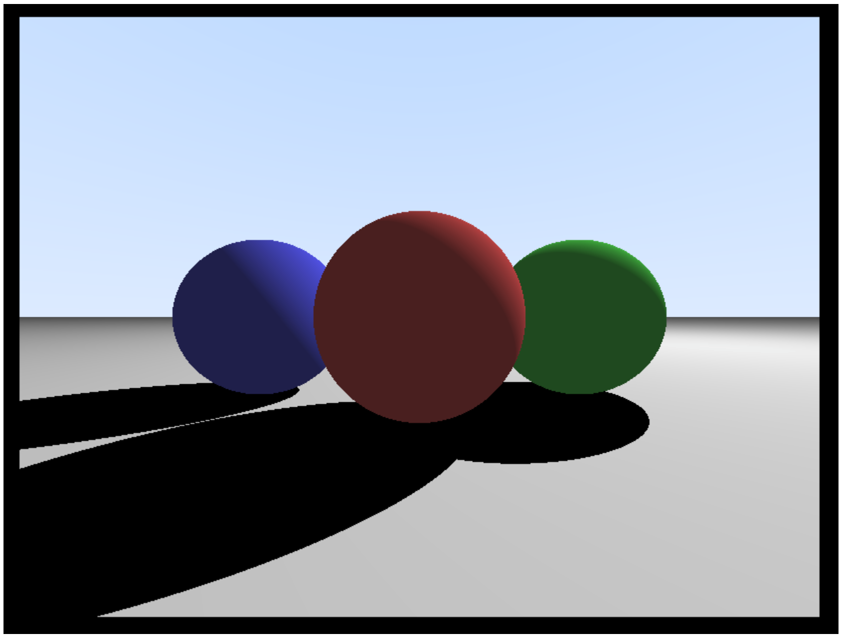
Part 2

```
0 Seconds, 255472 uSeconds.
```

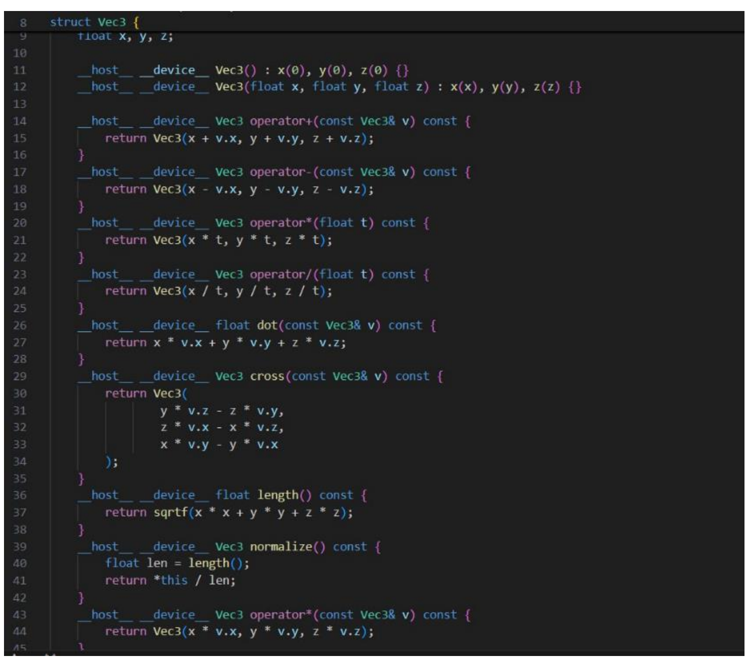
```
└─ nvprof ./main
==20086== NVPROF is profiling process 20086, command: ./main
0
1 Seconds, 1714834 uSeconds.
^C==20086== Profiling application: ./main
==20086== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   53.69%  300.33us         1  300.33us  300.33us  300.33us  [CUDA memcpy DtoH]
                   39.05%  218.44us         1  218.44us  218.44us  218.44us  gpuPart(unsigned char*, double*, int, int)
                    7.27%  40.641us         3  13.547us     576ns  39.457us  [CUDA memcpy HtoD]
      API calls:   96.65%  168.41ms         2  84.207ms  95.197us  168.32ms  cudaMalloc
                    1.90%  3.3134ms       114  29.065us      76ns  1.9025ms  cuDeviceGetAttribute
                    1.06%  1.8553ms         2  927.67us  116.69us  1.7387ms  cudaMemcpy
                    0.22%  391.57us         2  195.78us  5.0180us  386.55us  cudaMemcpyToSymbol
                    0.13%  219.97us         1  219.97us  219.97us  219.97us  cudaDeviceSynchronize
                    0.02%  35.286us         1  35.286us  35.286us  35.286us  cudaLaunchKernel
                    0.01%  17.127us         1  17.127us  17.127us  17.127us  cuDeviceGetName
                    0.00%  6.6830us         1  6.6830us  6.6830us  6.6830us  cuDeviceGetPCIBusId
                    0.00%  1.8480us         3     616ns      89ns  1.6430us  cuDeviceGetCount
                    0.00%     480ns         1     480ns     480ns     480ns  cuModuleGetLoadingMode
                    0.00%     383ns         2     191ns     103ns     280ns  cuDeviceGet
                    0.00%     248ns         1     248ns     248ns     248ns  cuDeviceTotalMem
                    0.00%     203ns         1     203ns     203ns     203ns  cuDeviceGetUuid
```

As you know, a large portion of the time is spent on **memory transfers**. The reason it takes longer to transfer from **device to host** is that the output is of type **double**, which occupies about **8 times more space** than uchar, making the transfer proportionally slower.

Part 3



In this method, for each pixel of the image, a **ray** is cast from the camera toward the scene. If the ray intersects an object, the color of that pixel is calculated based on the object's properties (such as color and light position). For each point where the ray intersects an object, a **shadow ray** is cast from that point toward the light source. If this shadow ray intersects another object, the point is in shadow and its color becomes darker. If the shadow ray does not intersect any object, the point is affected by **direct light**, and its color is calculated based on the light intensity.



```
struct Vec3 {
    float x, y, z;

    __host__ __device__ Vec3() : x(0), y(0), z(0) {}
    __host__ __device__ Vec3(float x, float y, float z) : x(x), y(y), z(z) {}

    __host__ __device__ Vec3 operator+(const Vec3& v) const {
        return Vec3(x + v.x, y + v.y, z + v.z);
    }
    __host__ __device__ Vec3 operator-(const Vec3& v) const {
        return Vec3(x - v.x, y - v.y, z - v.z);
    }
    __host__ __device__ Vec3 operator*(float t) const {
        return Vec3(x * t, y * t, z * t);
    }
    __host__ __device__ Vec3 operator/(float t) const {
        return Vec3(x / t, y / t, z / t);
    }
    __host__ __device__ float dot(const Vec3& v) const {
        return x * v.x + y * v.y + z * v.z;
    }
    __host__ __device__ Vec3 cross(const Vec3& v) const {
        return Vec3(
                y * v.z - z * v.y,
                z * v.x - x * v.z,
                x * v.y - y * v.x
        );
    }
    __host__ __device__ float length() const {
        return sqrtf(x * x + y * y + z * z);
    }
    __host__ __device__ Vec3 normalize() const {
        float len = length();
        return *this / len;
    }
    __host__ __device__ Vec3 operator*(const Vec3& v) const {
        return Vec3(x * v.x, y * v.y, z * v.z);
    }
```

Struct vec3:

This structure is used to represent **nodes and points** in three-dimensional space.

Mathematical operations such as **addition, subtraction, multiplication, and normalization** of the nodes are implemented in this structure.

```
struct Ray {
    Vec3 origin;
    Vec3 direction;

    __host__ __device__ Ray() {}
    __host__ __device__ Ray(const Vec3& o, const Vec3& d) : origin(o), direction(d) {}

    __host__ __device__ Vec3 at(float t) const {
        return origin + direction * t;
    }
};
```

```
struct Hittable {
    int type;          // Object type: SPHERE or PLANE
    Vec3 center;       // For sphere and plane (point on the plane)
    Vec3 normal;       // For plane normal (for PLANE type)
    float radius;      // For sphere
    Vec3 color;        // Material color
};
```

Ray Struct:

This structure represents a **ray**, which includes an **origin** and a **direction**.

struct Hittable:

This structure is used to represent **intersectable objects** (such as spheres and planes).

```
__device__ bool hitSphere(const Hittable& sphere, const Ray& r, float t_min, float t_max, float& t, Vec3& normal) {
    Vec3 oc = r.origin - sphere.center;
    float a = r.direction.dot(r.direction);
    float half_b = oc.dot(r.direction);
    float c = oc.dot(oc) - sphere.radius * sphere.radius;
    float discriminant = half_b * half_b - a * c;
    if (discriminant > 0) {
        float sqrt_d = sqrtf(discriminant);
        float root = (-half_b - sqrt_d) / a;
        if (root < t_max && root > t_min) {
            t = root;
            normal = (r.at(t) - sphere.center).normalize();
            return true;
        }
        root = (-half_b + sqrt_d) / a;
        if (root < t_max && root > t_min) {
            t = root;
            normal = (r.at(t) - sphere.center).normalize();
            return true;
        }
    }
    return false;
}
```

```
__device__ bool hitPlane(const Hittable& plane, const Ray& r, float t_min, float t_max, float& t, Vec3& normal) {
    float denom = plane.normal.dot(r.direction);
    if (fabsf(denom) > 1e-6f) { // Not parallel
        t = (plane.center - r.origin).dot(plane.normal) / denom;
        if (t < t_max && t > t_min) {
            normal = plane.normal;
            return true;
        }
    }
    return false;
}
```

functions hitSphere and hitPlane:

These functions check whether a **ray intersects** a sphere or a plane.

If an intersection occurs, the parameters **t** (the distance to the intersection) and **normal** (the surface normal at the intersection point) are calculated.

```cpp
if (hit_index >= 0) {
    Vec3 hit_point = r.at(closest_t);
    Vec3 light_dir = (light_pos - hit_point).normalize();
    float intensity = fmaxf(0.0f, normal.dot(light_dir));

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

    Vec3 result_color = ambient + diffuse;

    // Shadow check
    if (objects[hit_index].type == PLANE) {
        Ray shadow_ray(hit_point + 0.001f * normal, light_dir); // Offset to avoid self-intersection
        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);
            }

            if (hit) {
                result_color = Vec3(0.f, 0.f, 0.f);
                break;
            }
        }
    }

    return result_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
```

Functuin rayColor:

This function determines the **color of each pixel** based on ray-object intersections and shadow calculations.