ulm university universität
uulm

Institut für Numerische Mathematik

Dr. Andreas F. Borchert

7. Juli 2017
Blatt 10

Parallele Programmierung mit C++ (SS 2017)

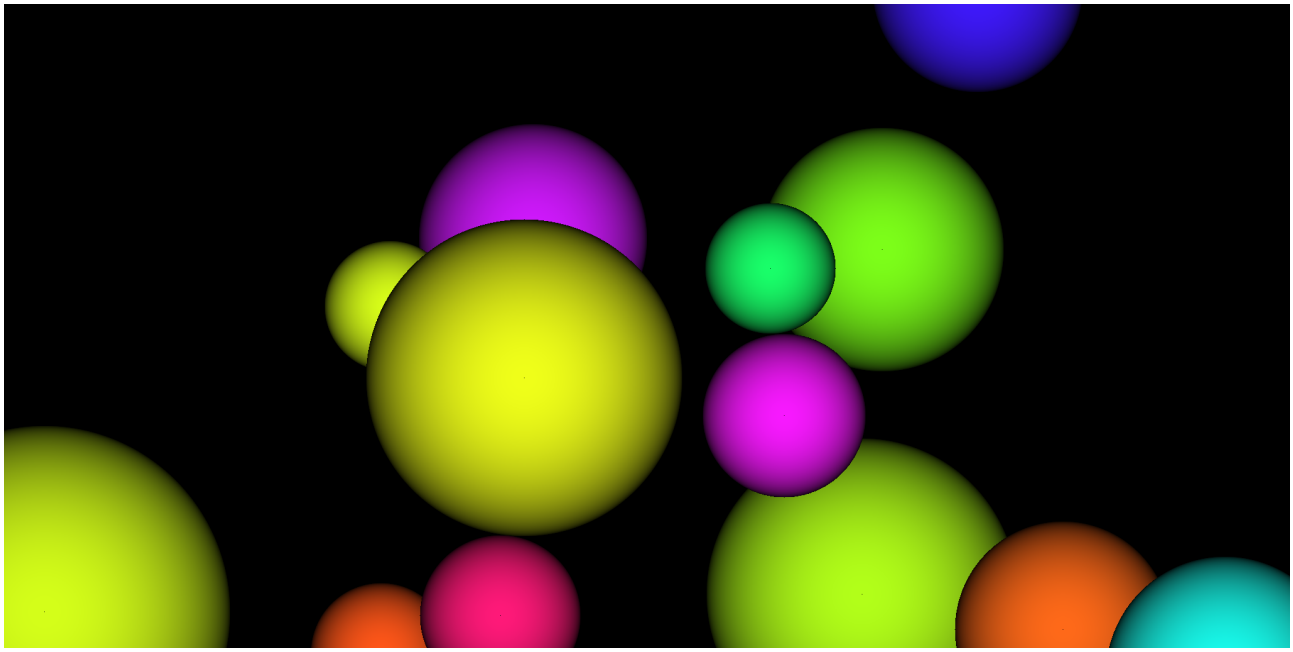
Abgabe bis zum 14. Juli 2017, 14:00 Uhr

Lernziele:

- Einfache Parallelisierung nach dem Fork-and-Join-Schema für eine GPU mit Hilfe von CUDA

Aufgabe 13: Bunte Kugeln

Ray-Tracing-Algorithmen eignen sich im besonderen Maße zur Parallelisierung, da jedes Pixel unabhängig von den anderen berechnet werden kann.



Um die Aufgabe zu erleichtern, steht Ihnen ein überaus vereinfachter nicht-parallelisierter Ray-Tracer zur Verfügung, der ausschließlich Kugeln unterstützt. Die Kugeln stellen die Methode *hit* zur Verfügung, der als Parameter ein Strahl (des Datentyps *Ray*) mitzugeben

ist, bestehend aus einem Punkt der zweidimensionalen Abbildungsfläche und einem Richtungsvektor.

```
struct Sphere {
    HOST_DEV Sphere() : center(0, 0, 0), radius(1), color(0, 0, 0) {
    }
    HOST_DEV Sphere(const Point& center,
                    double radius, RGBColor color) :
            center(center), radius(radius), color(color) {
    }
    HOST_DEV bool hit(const Ray& ray, double& tmin, Shade& shade) {
        /* ... */
    }
    Point center;
    double radius;
    RGBColor color;
};
```

Wenn der Strahl die Kugel trifft, dann wird in *tmin* der kürzeste Längenfaktor für den Richtungsvektor des Strahls zurückgegeben, mit dem ausgehend von dem Ausgangspunkt die Kugeloberfläche erreicht wird (das entspricht der Distanz zwischen dem Ausgangspunkt des Strahls und dem Schnittpunkt mit der Kugel, wenn der Richtungsvektor des Strahls die Länge 1 hat). Ferner finden sich dann in *shade* die Koordinaten des Schnittpunkts des Strahls mit der Kugel, der Normalenvektor (orthogonal zur Tangentialebene der Kugeloberfläche an dem Schnittpunkt) und die Farbe an der Oberfläche (*color*).

```
struct Shade {
    Point hit;
    Vector normal;
    RGBColor color;
};
```

Mit Hilfe des Skalarprodukts aus dem Richtungsvektor und des Normalenvektors lässt sich dann ein Farbübergang gestalten, der einer Beleuchtung aus der Betrachtungsrichtung entspricht. Für jeden Punkt der Abbildungsfläche wird der Strahl durch sämtliche Kugeln geschickt. Wenn mehrere Kugeln getroffen werden, dann wird die nächstgelegene Kugel in Betracht gezogen, da diese dann an diesem Punkt die dahinter liegenden Kugeln verdeckt.

```
HOST_DEV RGBColor get_color(Sphere* spheres, const Ray& ray) {
    Shade shade; double tmin; bool hit = false;
    for (unsigned int i = 0; i < SPHERES; ++i) {
        Shade last_shade; double last_tmin;
        if (spheres[i].hit(ray, last_tmin, last_shade)) {
            if (!hit || last_tmin < tmin) {
                tmin = last_tmin; shade = last_shade; hit = true;
            }
        }
    }
    if (!hit) return RGBColor(0, 0, 0);
```

```
    double factor = normalize(ray.direction) * normalize(shade.normal);
    if (factor >= 0) return RGBColor(0, 0, 0); /* unexpected */
    return shade.color * fabs(factor);
}
```

Die Vorlage ist bereits für die Verwendung auf einer GPU etwas vorbereitet. In der Header-Datei *cuda.hpp* sehen Sie eine Definition *HOST_DEV*, die von dem Präprozessor des *nvcc* in die Schlüsselwortkombination **__host__ __device__** übersetzt wird. Damit lassen sich Funktionen und Methoden kennzeichnen, die sowohl auf der CPU als auch der GPU sich aufrufen lassen. (Diese werden dann für jede der beiden Plattformen übersetzt.) Auf diese Weise lassen sich all die zur Verfügung gestellten Klassen und ihre Methoden sowohl auf der CPU als auch der GPU verwenden.

Im Rahmen der Aufgabe ist dann ausgehend von *tracer.cpp* die verschachtelte **for**-Schleifenkonstruktion, die sämtliche Pixel erzeugt, mit Hilfe einer Kernel-Funktion zu parallelisieren. Hier bietet es sich an, sowohl die Blöcke als auch das Gitter zweidimensional zu partitionieren.

Etwas trickreich gestaltet sich das *Makefile* für die CUDA-Fassung, da der *nvcc* nicht ohne weiteres die Optionen akzeptiert, die *pkg-config* zurückliefert. Beachten Sie bitte das beiliegende *Makefile.cuda*, das dieses Problem löst.

Wenn es zu Problemen kommen sollte, empfiehlt sich die Verwendung des Werkzeugs *cuda-memcheck*, dem Ihre CUDA-Anwendung mitsamt den Kommandozeilenargumenten zu übergeben ist. Alle Versuche, auf GPU-Speicher außerhalb der zugewiesenen Speicherflächen zuzugreifen, werden dann protokolliert.

Verpacken Sie all Ihre Quellen wiederum mit *tar* in ein Archiv und reichen Sie dies ein:

```
tar cvf tracer.tar tracer.cu *.?pp [mM]akefile
submit pp 13 tracer.tar
```

Viel Erfolg!