Universitatea Tehnică „Gheorghe Asachi” din Iași
Facultatea de Automatică și Calculatoare Domeniul:
Sisteme distribuite si tehnologii web

Ray Tracing în CUDA

PROIECT PROGRAMARE GPU

Coordonator științific:
assoc. prof. Simona Caraiman
Cîrstea Lucian-Cătălin

Student:
Mircea Mihai-Adrian

1. Introducere

1.1 Context și problemă

Ray tracing-ul este o tehnică de grafică computerizată utilizată pentru generarea de imagini realiste prin simularea comportamentului razelor de lumină într-un mediu tridimensional. Cu toate acestea, implementarea pe unitățile de procesare centrală (CPU) poate deveni ineficientă, mai ales în cazul scenelor complexe. În astfel de situații, numărul mare de raze de lumină și interacțiunile elaborate dintre acestea și obiectele 3D impun o cerință ridicată de putere computațională.

În cadrul problemei abordate, focusul se concentrează pe generarea imaginilor prin tehnica ray tracing în contextul graficii computerizate. Procesul implică urmărirea razei de lumină de la sursa de lumină către observator, iar aceasta interacționează cu obiectele din scenă prin intersectarea lor. De asemenea, se calculează culorile și umbrele în funcție de proprietățile materialelor și sursele de lumină, oferind o imagine detaliată și foto-realistică a scenei tridimensionale.

Pentru a realiza acest proces complex, ray tracing-ul utilizează principii precum reflexie, refracție, și difuzie pentru a simula comportamentul real al luminii. Este o tehnică esențială în domenii precum animație, efecte vizuale, design arhitectural și product design, contribuind la crearea unor vizualizări realiste și estetice.În concluzie, accelerația acestui algoritm prin implementarea sa pe unități de procesare grafică (GPU), în loc de unități de procesare centrală (CPU), devine crucială pentru gestionarea eficientă a complexității scenelor și pentru obținerea unor rezultate în timp util.

1.2 Componentele algoritmice care necesită accelerare

1.2.1 Generarea scenei

Implementarea funcției “create_world” pe GPU pentru a genera și inițializa hitables (obiectele 3D) și camera în scenă. Această funcție creează o scenă complexă cu sfere, materiale diverse și iluminare.

1.2.2 Calculul culorii

Funcția “color” este esențială pentru calculul culorilor în fiecare pixel. Aceasta parcurge razele și interacționează cu obiectele din scenă, aplicând modele de reflexie și refracție, și generând culorile corespunzătoare.

1.2.3 Randarea imaginii

Funcția render în acest context are rolul central în generarea imaginii finale prin aplicarea funcției color pentru fiecare rază de lumină asociată unui pixel din imaginea finală.

1.2.3.1 Anti-Aliasing

Funcția “render“ integrează tehnicile de anti-aliasing, un proces esențial în grafica computerizată pentru a reduce artefactele și a îmbunătăți calitatea generală a imaginilor generate.

Anti-aliasing-ul se realizează prin luarea unui număr de eșantioane pentru fiecare pixel. În loc să traseze o singură rază de lumină pentru fiecare pixel, se iau mai multe eșantioane într-o manieră controlată și se calculează media acestora pentru a obține culoarea finală a pixelului. Acest

proces contribuie la diminuarea efectului de "scădere a rezoluției" sau a contururilor neregulate în imagini.

1.2.3.2 Ray Tracing

Pentru fiecare pixel, se generează o rază de lumină corespunzătoare în funcție de parametrii camerei (cum ar fi poziția, direcția și unghiul de vedere). Această rază este apoi urmărită prin scenă, intersectând obiectele 3D și calculând culorile și umbrele corespunzătoare conform principiilor ray tracing-ului.

1.2.3.3 Acumularea culorilor

Culoarea calculată pentru fiecare eșantion este adunată la o variabilă de acumulare pentru pixelul respectiv. Procesul se repetă pentru toate eșantioanele asociate pixelului, iar apoi media acestora este calculată pentru a obține culoarea finală a pixelului.

```
__device__ vec3 color(const ray& r, hitable** world, curandState* local_rand_state) {
    ray cur_ray = r;
    vec3 cur_attenuation = vec3(1.0, 1.0, 1.0);
    int depth = 50;
    for (int i = 0; i < depth; i++) {
        hit_record rec;
        if ((*world)->hit(cur_ray, 0.001f, FLT_MAX, rec)) {
            ray scattered;
            vec3 attenuation;
            if (rec.mat_ptr->scatter(cur_ray, rec, attenuation, scattered, local_rand_state)) {
                cur_attenuation *= attenuation;
                cur_ray = scattered;
            }
            else {
                return vec3(0.0, 0.0, 0.0);
            }
        }
        else {
            vec3 unit_direction = unit_vector(cur_ray.direction);
            float t = 0.5f * (unit_direction.y() + 1.0f);
            vec3 c = (1.0f - t) * vec3(1.0, 1.0, 1.0) + t * vec3(0.5, 0.7, 1.0);
            return cur_attenuation * c;
        }
    }
}
```

Imaginea 1. Funția de calcul a culorii

După ce toți pixelii sunt procesați, imaginea finală este obținută prin combinarea culorilor calculate pentru fiecare pixel. Această imagine poate fi apoi salvată sau afișată, oferind rezultatul final al întregului proces de ray tracing.

Prin această abordare, funcția “render” îmbină concepte precum anti-aliasing, ray tracing și gestionarea culorilor pentru a produce o imagine finală care să reflecte iluminarea și proprietățile materialelor dintr-o scenă tridimensională.

1.3 Analiza complexității

1.3.1 Complexitatea operațiilor

Ray tracing-ul implică un număr semnificativ de operații complexe, cum ar fi intersecții ray-sferă, calculul culorilor și gestionarea umbrelor. Aceste operații au o complexitate individuală ridicată, iar numărul total de operații crește odată cu complexitatea scenei (numărul de obiecte, iluminarea, materialele).

1.3.2 Paralelizarea pe GPU

GPU-urile sunt dispozitive specializate pentru executarea eficientă a unui număr mare de operații în paralel, beneficiind de arhitecturi precum CUDA cores. Paralelizarea pe GPU aduce un avantaj semnificativ în contextul ray tracing-ului, permitând efectuarea simultană a mai multor intersecții și calculări de iluminare pentru diverse raze de lumină. Această metodă distribuie sarcinile de calcul pe un număr extins de unități de procesare, accelerând în mod substanțial procesul de generare a imaginilor.

1.3.3 Numărul de pixeli si eșantioane

Numărul de pixeli în imagine și numărul de eșantioane luate pentru fiecare pixel influențează direct complexitatea algoritmului. Cu cât există mai mulți pixeli sau eșantioane, cu atât crește numărul total de raze care trebuie urmărite și calculate.

Pe GPU, această creștere poate fi gestionată eficient prin împărțirea sarcinilor între diferite blocuri și fire de execuție, iar GPU-ul poate profita de paralelismul oferit de acestea.

1.3.4 Scalabilitate si optimizări

Paralelizarea pe GPU aduce beneficii semnificative în termeni de scalabilitate. Cu cât există mai multe nuclee pe GPU, cu atât mai multe operații pot fi efectuate simultan.

Optimizările specifice GPU-urilor, precum memoria partajată și cache-ul, pot fi exploatate pentru a minimiza timpul de acces la date și pentru a optimiza comunicarea între nucleele de procesare.

1.3.5 Memorie si transferul de date

Eficiența în gestionarea memoriei este crucială. Transferul eficient al datelor între CPU și GPU, precum și gestionarea memoriei GPU, influențează performanța generală. Utilizarea memoriei partajate sau cache-ului poate reduce timpul de acces și poate îmbunătăți eficiența memoriei, în special pentru datele care sunt frecvent utilizate.

1.3.6 Tipuri de obiecte si materiale

Complexitatea scenei, inclusiv tipurile de obiecte și materialele utilizate, afectează timpul de calcul. Materialele care necesită calcule mai complexe pentru iluminare sau umbre pot adăuga un nivel suplimentar de complexitate.

În ansamblu, paralelizarea pe GPU în cadrul algoritmului de ray tracing aduce un avantaj semnificativ în gestionarea complexității operațiunilor și permite o accelerare eficientă a generării imaginilor în comparație cu implementările pe CPU, mai ales în cazul scenelor mari și complexe.

1.4 Aplicații și produse pe piață

1.4.1 Produse și jocuri

Multe produse și jocuri de top utilizează ray tracing cu accelerare pe GPU pentru a oferi imagini realiste și detaliate. Exemple includ jocuri precum "Cyberpunk 2077" și "Minecraft" cu path tracing, precum și software de modelare 3D ca "Blender" sau "Autodesk Maya".

1.4.2 Industria filmului și designului

În industria filmului, ray tracing-ul pe GPU este utilizat pentru a crea efecte vizuale spectaculoase în filme precum "The Mandalorian". În domeniul designului, aplicații precum "KeyShot" beneficiază de accelerarea GPU-ului pentru randarea fotorealistică a modelelor.

2. Implementare

2.1 Abordare naivă / inițială

Abordarea naivă pentru paralelizarea ray tracing-ului pe GPU constă în a aloca fiecărui fir de execuție CUDA o sarcină independentă de calcul. Fiecare fir de execuție este responsabil pentru generarea culorilor unui pixel specific din imaginea finală, prin urmărirea unui număr de raze de lumină asociate acelui pixel. Această abordare simplă, dar eficientă în contextul scenei de test, implică divizarea imaginii în blocuri și fire de execuție CUDA.

2.1.1 Kernel-ul create_world

Kernel-ul create_world inițializează o scenă 3D cu sfere și alte obiecte folosind o abordare naivă, rulând pe un singur bloc și un fir de execuție CUDA. Datele necesare sunt alocate în memoria GPU folosind cudaMalloc, iar procesul implică:

- Crearea unui obiect sferă în centrul scenei și adăugarea de sfere mici în jurul acesteia, fiecare cu propriile caracteristici (poziție, rază, material).
- Materialele sunt atribuite aleator, iar alegerea tipului de material pentru fiecare sferă este influențată de un factor aleator.
- Parametrii camerei sunt setați pentru controlul perspectivelor.
- Alocarea datelor se face folosind cudaMalloc pentru a permite accesul concurent și eficient al fiecărui fir de execuție CUDA la structurile de date relevante.

```cpp
__global__ void create_world(hitable** d_list, hitable** d_world, camera** d_camera, int nx, int ny, curandState* rand_state) {
    if (threadIdx.x == 0 && blockIdx.x == 0) {
        curandState local_rand_state = *rand_state;
        d_list[0] = new sphere(vec3(0, -1000.0, -1), 1000,
            new lambertian(vec3(0.5, 0.5, 0.5)));
        int i = 1;
        for (int a = -11; a < 11; a++) {
            for (int b = -11; b < 11; b++) {
                float choose_mat = RND;
                vec3 center(a + RND, 0.2, b + RND);
                if (choose_mat < 0.8f) {
                    d_list[i++] = new sphere(center, 0.2,
                        new lambertian(vec3(RND * RND, RND * RND, RND * RND)));
                }
                else if (choose_mat < 0.95f) {
                    d_list[i++] = new sphere(center, 0.2,
                        new metal(vec3(0.5f * (1.0f + RND), 0.5f * (1.0f + RND), 0.5f * (1.0f + RND)), 0.5f * RND));
                }
                else {
                    d_list[i++] = new sphere(center, 0.2, new dielectric(1.5));
                }
            }
        }
        d_list[i++] = new sphere(vec3(0, 1, 0), 1.0, new dielectric(1.5));
        d_list[i++] = new sphere(vec3(-4, 1, 0), 1.0, new lambertian(vec3(0.4, 0.2, 0.1)));
        d_list[i++] = new sphere(vec3(4, 1, 0), 1.0, new metal(vec3(0.7, 0.6, 0.5), 0.0));
        *rand_state = local_rand_state;
        *d_world = new hitable_list(d_list, 22 * 22 + 1 + 3);

        vec3 lookfrom(13, 2, 3);
        vec3 lookat(0, 0, 0);
        float dist_to_focus = 10.0; (lookfrom - lookat).length();
        float aperture = 0.1;
        *d_camera = new camera(lookfrom,
            lookat,
            vec3(0, 1, 0),
            30.0,
            float(nx) / float(ny),
            aperture,
            dist_to_focus);
    }
}
```

Imaginea 2. Kernel-ul create_world

2.1.2 Kernel-ul render_init

Kernel-ul de inițializare alocă pentru fiecare pixel un stadiu de generație a numerelor aleatoare. Această inițializare a numerelor aleatoare este esențială pentru diversificarea direcțiilor razelor de lumină în cadrul scenei.

```cpp
__global__ void render_init(int max_x, int max_y, curandState* rand_state) {
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    int j = threadIdx.y + blockIdx.y * blockDim.y;
    if ((i >= max_x) || (j >= max_y)) return;
    int pixel_index = j * max_x + i;
    curand_init(2024 + pixel_index, 0, 0, &rand_state[pixel_index]);
}
```

Imaginea 3. Kernel-ul render_init

2.1.3 Kernel-ul render

Kernel-ul render primește parametri precum dimensiunile imaginii, numărul de eșantioane per pixel și o referință la camera și scenă. Fiecare fir de execuție CUDA este responsabil pentru un pixel al imaginii și efectuează un buclu pentru a estima culoarea pixelului prin urmărirea mai multor raze de lumină. Coordonatele pixelilor și parametrii de eșantionare sunt utilizați pentru a genera razele de lumină asociate. Culorile estimate pentru fiecare rază sunt acumulate, iar rezultatul este normalizat la final și stocat în buffer-ul de imagine, reprezentat de vectorul vec3* fb.

Este important de menționat că fb este alocat în memoria GPU (Graphic Processing Unit) și, după finalizarea kernel-ului, acesta va fi transferat în memoria CPU (Central Processing Unit) pentru a putea fi utilizat ulterior. Acest proces de transfer poate fi realizat prin intermediul funcției cudaMemcpy pentru a asigura sincronizarea și accesul corect la rezultatele calculate pe dispozitivul GPU.

```
__global__ void render(vec3* fb, int max_x, int max_y, int ns,
    camera** cam, hitable** world, curandState* rand_state) {
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    int j = threadIdx.y + blockIdx.y * blockDim.y;
    if ((i >= max_x) || (j >= max_y)) return;
    int pixel_index = j * max_x + i;
    curandState local_rand_state = rand_state[pixel_index];
    vec3 col(0, 0, 0);
    for (int s = 0; s < ns; s++) {
        float u = float(i + curand_uniform(&local_rand_state)) / float(max_x);
        float v = float(j + curand_uniform(&local_rand_state)) / float(max_y);
        ray r = (*cam)->get_ray(u, v, &local_rand_state);
        col += color(r, world, &local_rand_state);
    }
    rand_state[pixel_index] = local_rand_state;
    col /= float(ns);
    col[0] = sqrt(col[0]);
    col[1] = sqrt(col[1]);
    col[2] = sqrt(col[2]);
    fb[pixel_index] = col;
}
```

Imaginea 4. Kernel-ul render

2.1.4 Kernel-ul free_world

Kernel-ul de eliberare a memoriei pentru obiectele create în scenă este, de asemenea, simplu și rulează pe un singur bloc și fir de execuție CUDA.

```
__global__ void free_world(hitable** d_list, hitable** d_world, camera** d_camera) {
    for (int i = 0; i < 22 * 22 + 1 + 3; i++) {
        delete ((sphere*)d_list[i])->mat_ptr;
        delete d_list[i];
    }
    delete* d_world;
    delete* d_camera;
}
```

Imaginea 5. Kernel-ul free_world

3. Rezultate experimentale

3.1 Detaliile dispozitivelor hardware / software utilizate

Detaliile dispozitivelor hardware/software utilizate în experimente sunt următoarele:

Nume dispozitiv	Valoare
CPU	Ryzen 7 5800X
GPU	RTX 2070 SUPER
OS	Windows 11

Tabel 1. Dispozitive hardware / software utilizate

Specificații obținute prin rularea aplicației deviceQuery.cpp din pachetul Cuda

Nume proprietate	Valoare
CUDA Driver Version / Runtime Version	12.3 / 12.3
CUDA Capability Major / Minor Version	7.5
Multiprocessors	2560 CUDA Cores
Max number of threads per multiprocessor	1024
Max number of threads per block	1024
Memory Bus Width	256-bit

Tabel 2. Specificatii CUDA

3.2 Setul de date utilizat

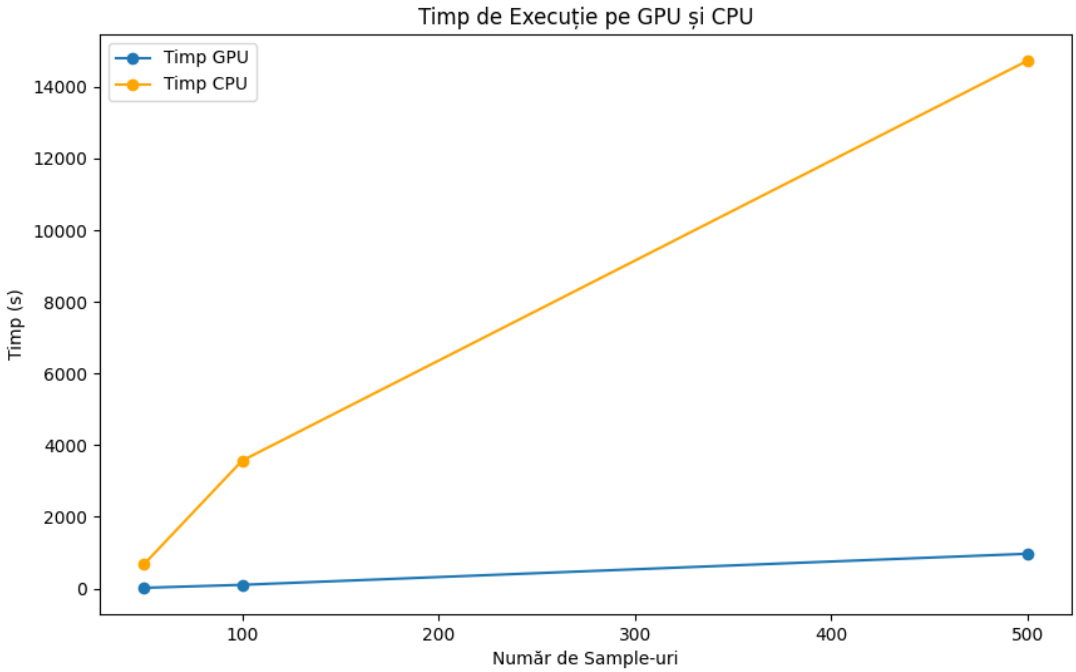
Seturile de date utilizate în experimente sunt caracterizate de diferite configurații de rezoluție și complexitate. În primul set, imaginea are dimensiunile 500x300 pixeli, cu 50 de eșantioane per pixel. Timpul de execuție pe GPU pentru acest set este de 18.843 secunde, în timp ce pe CPU atinge 674.412 secunde.

Al doilea set de date implică o imagine mai detaliată, cu 800x600 pixeli și 100 de eșantioane per pixel. Performanța GPU-ului în această configurație este notabilă, cu un timp de execuție de 102.072 secunde, comparativ cu 3567.129 secunde pe CPU.

Cel de-al treilea set de date reprezintă un scenariu cu o rezoluție superioară, având imaginea dimensiunile 1200x800 pixeli și 500 de eșantioane per pixel. GPU-ul continuă să ofere performanță remarcabilă, realizând procesarea în 969.351 secunde, în timp ce CPU-ul înregistrează un timp de execuție semnificativ mai mare, atingând 14730.921 secunde. Aceste seturi de date reflectă diversitatea condițiilor testate și evidențiază eficiența soluției paralele pe GPU în comparație cu abordarea secvențială pe CPU.

Set de date	Width	Heigth	Numar sample	Timp GPU (s)	Timp CPU (s)
Setul 1	500	300	50	18.843	674.412
Setul 2	800	600	100	102.072	3567.129
Setul 3	1200	800	500	969.351	42730.921

Tabel 3. Setul de date utilizat



Imaginea 6. Compararea performanțelor de execuție GPU vs. CPU în funcție de dimensiunea setului de date

3.3 Analiza rezultatelor

3.3.1 Timpul de rulare dedicat transferului datelor

Pentru setul 1 de date în care imaginea are 500x300 pixeli, cu 50 de eșantioane per pixel, timpul de copiere a vectorului tridimensional rezultat de pe device pe host este de 282us. În implementarea curentă, copierea de pe host pe device nu este relevantă, șcena fii creată în memoria globală.

cudaMemcpy	1	282us	282us	282us	282us	0ns
cudaMalloc	6	194.134ms	32.3557ms	14.1us	193.926ms	72.2564ms
cudaLaunchKernel	4	2.99954s	749.886ms	677.454ms	887.063ms	82.272ms
cudaGetLastError	4	5.7us	1.425us	900ns	2.5us	653.357ns
cudaDeviceSynchronize	5	19.1953s	3.83906s	27.7us	19.1644s	7.66266s
cuModuleGetLoadingMode	1	2us	2us	2us	2us	0ns
cuModuleGetFunction	5	2.0215ms	404.3us	220.3us	690.4us	202.945us
cuMemcpyDtoH_v2	1	254.7us	254.7us	254.7us	254.7us	0ns
cuMemAlloc_v2	6	358.1us	59.6833us	9.2us	184us	65.8434us
cuLibraryLoadData	1	15.2276ms	15.2276ms	15.2276ms	15.2276ms	0ns
cuLibraryGetModule	1	2.9001ms	2.9001ms	2.9001ms	2.9001ms	0ns
cuLaunchKernel	4	2.97936s	744.839ms	658.959ms	886.312ms	86.3613ms
cuInit	1	57.1032ms	57.1032ms	57.1032ms	57.1032ms	0ns
cuGetProcAddress_v2	432	103.8us	240.278ns	0ns	13.2us	721.911ns
cuDriverGetVersion	1	1.7us	1.7us	1.7us	1.7us	0ns
cuDeviceTotalMem_v2	1	400ns	400ns	400ns	400ns	0ns
cuDevicePrimaryCtxRetain	1	193.592ms	193.592ms	193.592ms	193.592ms	0ns
cuDeviceGetUuid	1	200ns	200ns	200ns	200ns	0ns
cuDeviceGetName	1	700ns	700ns	700ns	700ns	0ns
cuDeviceGetLuid	1	400ns	400ns	400ns	400ns	0ns
cuDeviceGetCount	1	400ns	400ns	400ns	400ns	0ns
cuDeviceGetAttribute	111	13.4us	120.721ns	0ns	1.9us	231.765ns
cuDeviceGet	1	1.9us	1.9us	1.9us	1.9us	0ns
cuCtxSynchronize	5	19.1952s	3.83904s	7.5us	19.1643s	7.66266s
cuCtxSetCurrent	1	800ns	800ns	800ns	800ns	0ns
cuCtxPushCurrent_v2	1	800ns	800ns	800ns	800ns	0ns
cuCtxPopCurrent_v2	1	1.7us	1.7us	1.7us	1.7us	0ns
cuCtxGetDevice	1	200ns	200ns	200ns	200ns	0ns
cuCtxGetCurrent	2	2.1us	1.05us	400ns	1.7us	650ns

Imaginea 7. Valorile din NSight pentru primul set de date.

Pentru setul 2 de date în care imaginea are 800x600 pixeli, cu 100 de eșantioane per pixel, timpul de copiere a vectorului tridimensional rezultat de pe device pe host este de 687.7us.

Name	Number of Calls	Total Duration	Average Duration	Minimum Duration	Maximum Duration	Duration Standard Deviation
cudaMemcpy	1	687.7us	687.7us	687.7us	687.7us	0ns
cudaMalloc	6	185.492ms	30.9154ms	16.5us	185.012ms	68.9141ms
cudaLaunchKernel	4	6.29233s	1.57308s	845.374ms	3.46907s	1.09976s
cudaGetLastError	4	5.7us	1.425us	800ns	3.2us	1.02561us
cudaDeviceSynchronize	5	10.525s	2.10501s	25.1us	10.4939s	4.19448s
cuModuleGetLoadingMode	1	2us	2us	2us	2us	0ns
cuModuleGetFunction	5	2.96ms	592us	200.7us	1.5406ms	499.751us
cuMemcpyDtoH_v2	1	659.8us	659.8us	659.8us	659.8us	0ns
cuMemAlloc_v2	6	621.7us	103.617us	11us	211.9us	75.0662us
cuLibraryLoadData	1	14.9126ms	14.9126ms	14.9126ms	14.9126ms	0ns
cuLibraryGetModule	1	3.0304ms	3.0304ms	3.0304ms	3.0304ms	0ns
cuLaunchKernel	4	6.27145s	1.56786s	827.039ms	3.46748s	1.10225s
cuInit	1	53.9383ms	53.9383ms	53.9383ms	53.9383ms	0ns
cuGetProcAddress_v2	432	108.9us	252.083ns	0ns	13us	717.761ns
cuDriverGetVersion	1	1.8us	1.8us	1.8us	1.8us	0ns
cuDeviceTotalMem_v2	1	700ns	700ns	700ns	700ns	0ns
cuDevicePrimaryCtxRetain	1	184.687ms	184.687ms	184.687ms	184.687ms	0ns
cuDeviceGetUuid	1	200ns	200ns	200ns	200ns	0ns
cuDeviceGetName	1	700ns	700ns	700ns	700ns	0ns
cuDeviceGetLuid	1	300ns	300ns	300ns	300ns	0ns
cuDeviceGetCount	1	400ns	400ns	400ns	400ns	0ns
cuDeviceGetAttribute	111	13.8us	124.324ns	0ns	2.1us	241.324ns
cuDeviceGet	1	2us	2us	2us	2us	0ns
cuCtxSynchronize	5	10.5249s	2.10499s	7.2us	10.4939s	4.19447s
cuCtxSetCurrent	1	800ns	800ns	800ns	800ns	0ns
cuCtxPushCurrent_v2	1	900ns	900ns	900ns	900ns	0ns
cuCtxPopCurrent_v2	1	1.6us	1.6us	1.6us	1.6us	0ns
cuCtxGetDevice	1	200ns	200ns	200ns	200ns	0ns
cuCtxGetCurrent	2	2.1us	1.05us	300ns	1.8us	750ns

Imaginea 8. Valorile din NSight pentru al doilea set de date.

Pentru setul 3 de date în care imaginea are 1200x800 pixeli, cu 500 de eșantioane per pixel, timpul de copiere a vectorului tridimensional rezultat de pe device pe host este de 687.7us.

Name	Number of Calls	Total Duration	Average Duration	Minimum Duration	Maximum Duration	Duration Standard Deviation
cudaMemcpy	1	1.0723ms	1.0723ms	1.0723ms	1.0723ms	0ns
cudaMalloc	6	189.06ms	31.5101ms	14us	188.394ms	70.161ms
cudaLaunchKernel	4	2.86038s	715.096ms	608.097ms	830.512ms	84.8871ms
cudaGetLastError	4	4.5us	1.125us	600ns	2.4us	739.51ns
cudaDeviceSynchronize	5	19.7277s	3.94554s	25.6us	19.6963s	7.87538s
cuModuleGetLoadingMode	1	2us	2us	2us	2us	0ns
cuModuleGetFunction	5	1.9425ms	388.5us	184.7us	664.5us	198.539us
cuMemcpyDtoH_v2	1	1.0438ms	1.0438ms	1.0438ms	1.0438ms	0ns
cuMemAlloc_v2	6	816us	136us	8.9us	377.2us	134.335us
cuLibraryLoadData	1	15.0695ms	15.0695ms	15.0695ms	15.0695ms	0ns
cuLibraryGetModule	1	2.8474ms	2.8474ms	2.8474ms	2.8474ms	0ns
cuLaunchKernel	4	2.84057s	710.142ms	607.39ms	829.883ms	83.08ms
cuInit	1	54.6591ms	54.6591ms	54.6591ms	54.6591ms	0ns
cuGetProcAddress_v2	432	111.2us	257.407ns	0ns	13.7us	743.329ns
cuDriverGetVersion	1	1.8us	1.8us	1.8us	1.8us	0ns
cuDeviceTotalMem_v2	1	500ns	500ns	500ns	500ns	0ns
cuDevicePrimaryCtxRetain	1	188.068ms	188.068ms	188.068ms	188.068ms	0ns
cuDeviceGetUuid	1	200ns	200ns	200ns	200ns	0ns
cuDeviceGetName	1	700ns	700ns	700ns	700ns	0ns
cuDeviceGetLuid	1	500ns	500ns	500ns	500ns	0ns
cuDeviceGetCount	1	500ns	500ns	500ns	500ns	0ns
cuDeviceGetAttribute	111	14.3us	128.829ns	0ns	1.9us	229.134ns
cuDeviceGet	1	1.9us	1.9us	1.9us	1.9us	0ns
cuCtxSynchronize	5	19.7276s	3.94553s	7.4us	19.6963s	7.87537s
cuCtxSetCurrent	1	700ns	700ns	700ns	700ns	0ns
cuCtxPushCurrent_v2	1	900ns	900ns	900ns	900ns	0ns
cuCtxPopCurrent_v2	1	1.8us	1.8us	1.8us	1.8us	0ns
cuCtxGetDevice	1	300ns	300ns	300ns	300ns	0ns
cuCtxGetCurrent	2	2.1us	1.05us	300ns	1.8us	750ns

Imaginea 9. Valorile din NSight pentru al treilea set de date.

3.3.2 Estimarea accelerării implementării CUDA

Pentru a evalua eficiența implementărilor CUDA, vom analiza speedup-ul obținut în comparație cu implementarea secvențială pe CPU. Speedup-ul este calculat utilizând formula:

$$Speedup = \frac{T_{GPU}}{T_{CPU}}$$

unde T_{CPU} reprezintă timpul de execuție pe CPU, iar T_{GPU} reprezintă timpul de execuție pe GPU.

Setul de date 1:

- Pixeli pe X: 500
- Pixeli pe Y: 300
- Număr de Sample-uri: 50
- Timp de Execuție pe CPU: 674.412s
- Timp de Execuție pe GPU: 18.843s

$$Speedup: \frac{674.412}{18.843} \approx 35.78$$

Setul de date 2:

- Pixeli pe X: 800
- Pixeli pe Y: 600
- Număr de Sample-uri: 100
- Timp de Execuție pe CPU: 3567.129s
- Timp de Execuție pe GPU: 102.072s

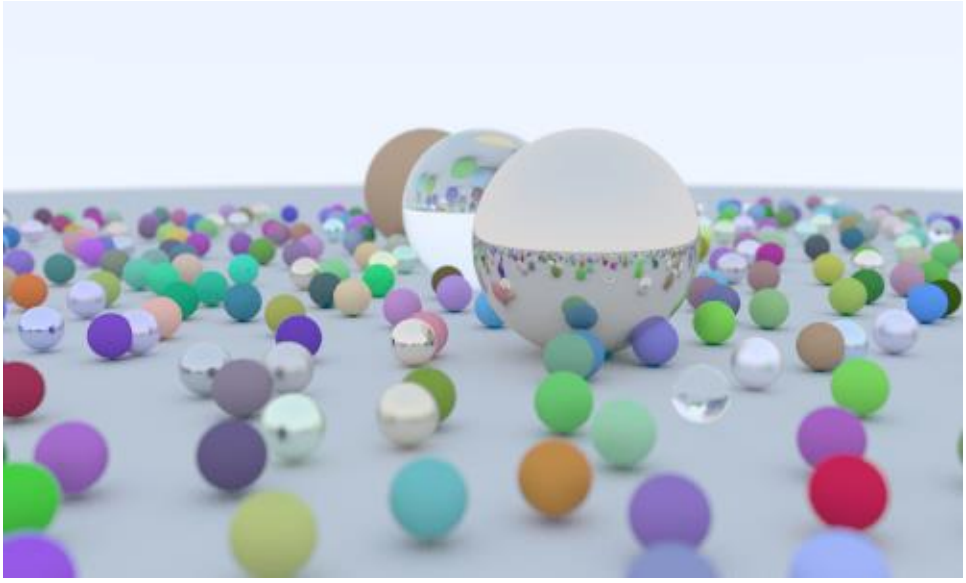
$$Speedup: \frac{3567.129}{102.072} \approx 34.96$$

Setul de date 3:

- Pixeli pe X: 1200
- Pixeli pe Y: 800
- Număr de Sample-uri: 500
- Timp de Execuție pe CPU: 14730.921s
- Timp de Execuție pe GPU: 969.351s

$$Speedup: \frac{14730.921}{969.351} \approx 15.20$$

Imagini



Imaginea 10. Imaginea finala pentru setul 3 de date.

Bibliografie

1. Pharr, M., & Jakob, W. (2016). Physically Based Rendering: From Theory to Implementation (3rd ed.) - https://www.pbr-book.org/
2. Shirley, P., & Morley, K. (2003). Realistic Ray Tracing (2nd ed.). Natick - https://books.google.ro/books?id=knpN6mnhJ8QC&printsec=frontcover&hl=ro&source=gbs_ge_summary_r&cad=0#v=onepage&q&f=false
3. CUDA Toolkit Documentation - https://docs.nvidia.com/cuda/
4. Nvidia DLSS & GeForce RTX - https://www.nvidia.com/en-us/geforce/news/nvidia-rtx-games-engines-apps/