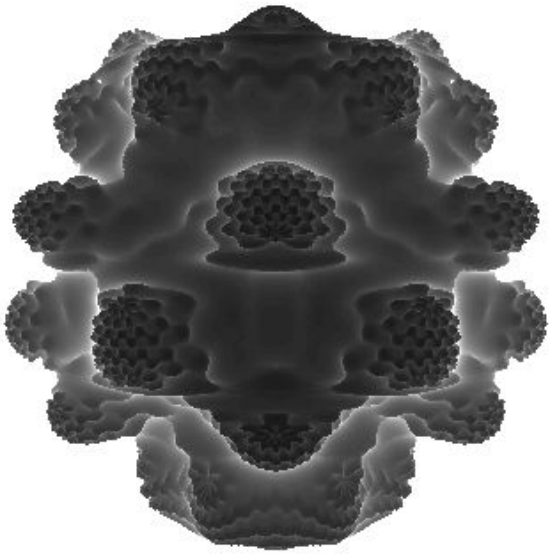
Курсов проект

по Системи за паралелна обработка

Тема: “Манделбълб”

Софийски университет "Св. Климент Охридски"

2017



Изготвил
Илия Сотиров Жечев, Ф.Н. 81125, Компютърни науки

Научен ръководител
ас. Христо Христов

Дата: Проверка: .....

Съдържание

Цел на проекта

Целта на проекта е да се разучат различни техники за паралелно генериране на визуализация на триизмерния фрактал “Манделбълб”, като се избере и реализира най-добрата от тях. За избраната за имплементация техника трябва да се направят замервания на различни показатели.

Изисквания към проекта:

- Реализацията на проекта трябва да предоставя интерфейс на потребителя през който може да се конфигурира генерирането на изображението.
- Програмата трябва да използва паралелни процеси за да се използва максимално изчислителната сила на машината на която тя се изпълнява и за да се ускори генерирането на крайното изображение.
- Програмата трябва да извежда съобщения за уведомяване на потребителя състоянието си и процеса на работа.
- Преди завършване програмата трябва да информира потребителя за отнетото време за генериране на изображението.

Описание на избрания алгоритъма

За да разберем как може да се подходи към проблема трябва да разберем какво представлява самия фрактал. Манделбълб подобно на двуизмерния си братовчед манделброт представлява множество от точки, в този случай триизмерни, които при итерация на определена функция НЕ дивергират.

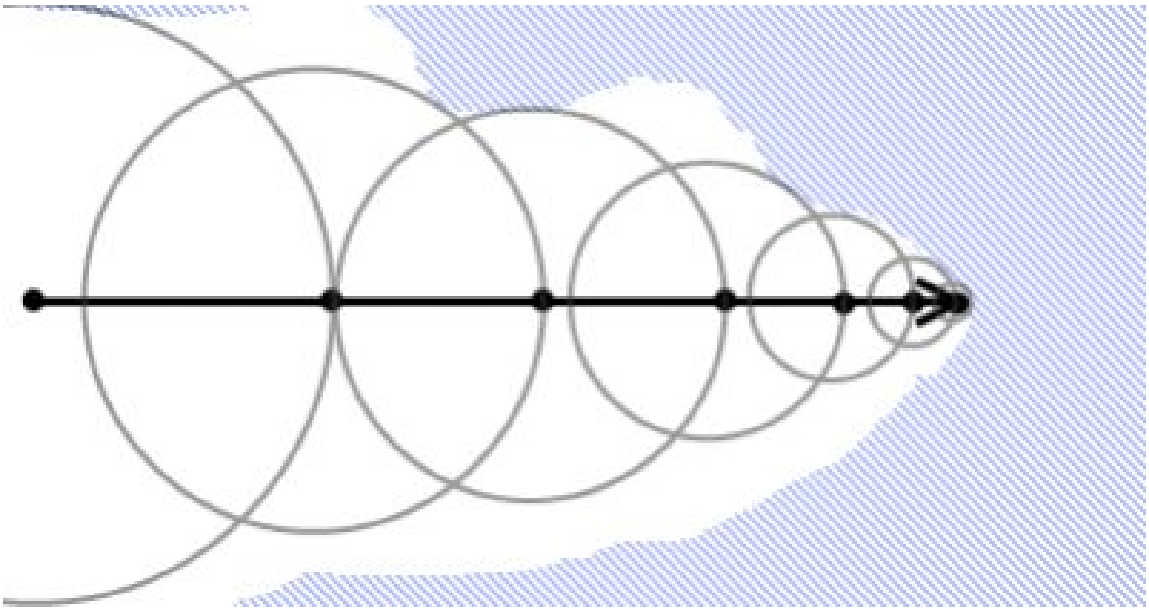
Съществуват много начини за подход към генерирането на изображение на фрактала. Тук ще споменем някой от тях и техните предимства и недостатъци.

➔ Итерация в триизмерен обем - този подход се характеризира като най-интуитивен и най-бавен. Избираме кубичен обем в пространството и го дискретизираме. За всяка точка (обем) итерираме нашата функция до определен предварително зададен лимит и спрямо това дали приключваме при дивергиране (вектора на итерация става прекалено голям) или стигаме лимита на итерациите избираме или отхвърляме точката на итерация.
Правейки това за всяка точка ни създава “облак” от точки които са от фрактала (тези които не са дивергирали). След това може да използваме различни графични библиотеки за визуализация на паралелно генерирания облак от точките на фрактала.
Проблема на този метод е че правим прекалено много итерации а накрая визуализираме само една проекция от тях. Голяма част от изчисленията са

ненужни за самата визуализация. Също така генерираното изображение често не изглежда толкова добре колкото изображения създадени чрез други подходи.

- ➔ Маршируване с лъчи - този метод може да се раздели на два основни типа.
  - ◆ Равномерно маршируване - генерирането при този подход става като “изстрелваме” лъч за всеки пиксел на изображението което ще генерираме и малко по малко предвижваме началото на този лъч в негова посока. На всяка стъпква проверяваме дали сме във фрактала по познатия вече начин. Проблема тук е че отново правим голям брой итерации и крайното изображение зависи от стъпата която правим с лъча. Колкото по малка е стъпката толкова е по точно изображението, но и по бавно се генерира.
  - ◆ Маршируване с функция на дистанцията - този подход се характеризира като най-бърз и точен. Тук отново “изстрелваме” лъчи по посока на фрактала, но стъпката, която прави всеки път, зависи от функция която ни дава най-близкото разстояние от началото на итерирания лъч до повърхността на фрактала. Тази функция се моделира на база на производната на функцията на итерацията.
    Използвайки този метод на всяко придвижване на лъча може да вземам максималната възможна стъпка без да влезем във фрактала.

Избирайки последния метод си гарнтираме минимален брой итерации и възможно най-точно изображение.
Отдолу може да видим интуитивно, защо и как става самото “маршируване” на един лъч.
Генерирането на изображението може да се паралеризира като го разбием на блокове (групи) за които да изберем броя на нишките които ще работят върху самите групи.



Формулите които използваме за генериране на фрактала са:

$$z_{n+1} = z_n^2 + c$$

Формула която вече е позната на хората генерирали манделброт.

$$DE = 0.5 * ln(r) * r/dr$$

Където

$$r = |f_n(c)| \text{ and } dr = |f_n'(c)|$$

Реализация

Езика на който ще реализираме програмата е Cuda/C++. Използвайки платформата която Cuda ни предлага може да се възползваме от силната паралеризация която ни предоставя видео картата на машината на която ще изпълняваме програмата си.

Всяка Cuda програма се състои от функции анотирани с __global__ или __device__, като характерното за тах е че се изпълняват на видео картата. Може да извикаме такава функция по начин с който да спцифицираме броя пъти който да се изпълни и броя на нишките които да се изпълнят за група от изпълняващи се ядра.

```
if(!test)
        printf("Starting kernel execution...\n");
d_main<<<block_dim, group_dim>>>(d_screen_buff, width, height);
if(!test)
        printf("Kernel execution ended.\n");
```

В случая изпълняваме функцията d_main като block_dim ни задава броя на блоковете, а group_dim ни задава броя на пикселите в група. Тези две променливи са триизмерни, но ние използваме само две от измеренията.

```
__global__ void d_main(
        pixel* screen_buffer,
        const size_t width,
        const size_t height
) {
        size_t x = (blockIdx.x * blockDim.x) + threadIdx.x;
        size_t y = (blockIdx.y * blockDim.y) + threadIdx.y;

        if(x < width && y < height) {
                float min_w_h = (float) min(width, height);

                float ar = (float) width / (float) height;
                float u = (float) x / min_w_h - ar * 0.5f;
                float v = (float) y / min_w_h - 0.5f;

                ray r = get_ray(u, v);
                float c = march(r) * 255.0f;
                float3 color = make_float3(c, c, c);

                screen_buffer[y * width + x] = color;
        }
}
```

Разглеждайки самата функция виждаме интерфейса който Cuda ни предоставя за идентифициране на пиксела който трябва да “оцветим” във всяко едно изпълнение на ядро.

След изпълнението на функцията трябва да копираме screen_buff който съдържа информацията за пикселите на host устройството и да запишем съответното изображение във файл.

```
if(!test)
        printf("Reading screan buffer from device...\n");
check_result(cudaMemcpy(h_screen_buff, d_screen_buff, num_pixels * sizeof(pixel), cudaMemcpyDeviceToHost));
if(!test)
        printf("Done.\n");

printf("Time taken (ms): %i\n", (int) ((double) (clock() - t_start) / CLOCKS_PER_SEC * 1000.0f));

if(!test){
        printf("Writing to file...\n");
        write_image(file_name, h_screen_buff, width, height);
        printf("Done\n");
}
```

За самото изчисление на цвета на всеки пиксел извършваме вече споменатото “маршируване” към фрактала като можем да конфигурираме различни параметри на маршируването за да получим различни по финност и скорост резултати.

```
__device__ float march(ray r) {
        float total_dist = 0.0;
        int max_ray_steps = 64;
        float min_distance = 0.002;

        int steps;
        for (steps = 0; steps < max_ray_steps; ++steps) {
                float3 p = r.o + r.d * total_dist;
                float distance = mandelbulb_de(p);
                total_dist += distance;
                if (distance < min_distance) break;
        }
        return 1.0 - (float) steps / (float) max_ray_steps;
}
```

Самата функция ни връща интерполация на броя на стъпките които са и отнели за да достигне до фрактала или да дивергира. Използвайки директно резултата от тази функция получаваме много добри резултати от гледна точка на оцветяване.

Резултати и тестове

В тази част на доклада ще изпълним програмата с различен брой нишки, ще направим замервания на времето за изпълнение, ще изчислим полезни статистики за различните параметри и ще генерираме графики за лесно сравнение на резултатите.

В таблиците по долу може да видим информация за замерванията на времето спрямо размерите на блоковете както и техните ускорения и ефективност.

X1	Y1	Time 1	X2	Y2	Time 2	X3	Y3	Time 3
1	1	15090	1	2	7156	2	4	1910
2	1	7433	3	2	2510	3	4	1270
4	1	3713	4	2	1866	5	4	756
6	1	2463	5	2	1463	6	4	640
8	1	1853	6	2	1250	7	4	540
10	1	1480	7	2	1070	8	4	480
12	1	1220	8	2	913	9	4	840
14	1	1063	9	2	833	10	4	750
16	1	936	10	2	753	11	4	666
18	1	833	11	2	690			

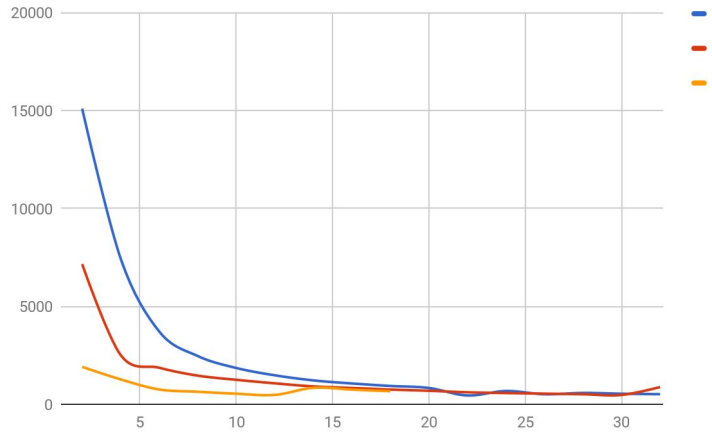
20	1	450	12	2	613			
22	1	680	13	2	573			
24	1	513	14	2	543			
26	1	580	15	2	513			
28	1	540	16	2	476			
30	1	516	17	2	880			
32	1	473	18	2	833			

Забелязваме че най добро време получваме за размер на блок - (31, 1) - 473ms

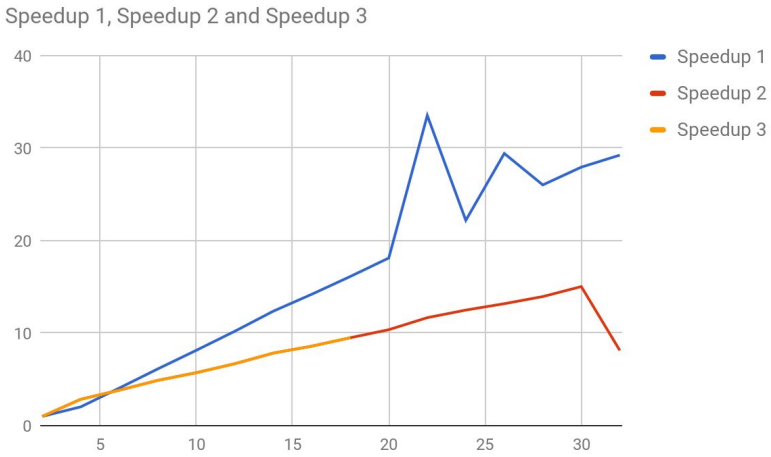
Speedup 1	Efficiency 1	Speedup 2	Efficiency 2	Speedup 3	Efficiency 3
1	1	1	0.5	1	0.125
2.03	0.51	2.85	0.475	2.85	0.2375
4.06	0.25	3.83	0.47875	3.83	0.1915
6.13	0.17	4.89	0.489	4.89	0.20375
8.14	0.13	5.72	0.47667	5.72	0.20429
10.2	0.1	6.69	0.47786	6.69	0.20906
12.37	0.09	7.84	0.49	7.84	0.21778
14.2	0.07	8.59	0.47722	8.59	0.21475
16.12	0.06	9.5	0.475	9.5	0.21591
18.12	0.06	10.37	0.47136		
33.53	0.08	11.67	0.48625		
22.19	0.05	12.49	0.48038		
29.42	0.05	13.18	0.47071		
26.02	0.04	13.95	0.465		
27.94	0.04	15.03	0.46969		
29.24	0.03	8.13	0.23912		
31.9	0.03	8.59	0.23861		

Ускорението смятаме по формулата T1/Tn, като T1 ни е времето за изпълнение на програмата на едно ядро. Ефективността смятаме по формулата: Sp/p, като Sp е ускорението а p броя на нишки.

Графика на времето спрямо броя нишки.

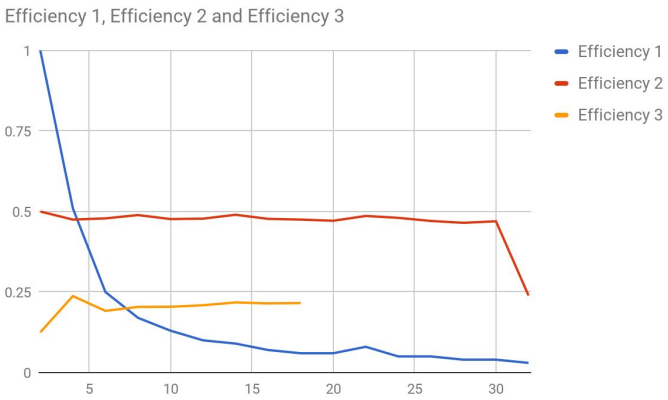


Графика на ускорението спрямо броя нишки.



На горната графика виждаме че ускорението на второто и третото измерване са почти идентини.

Графика на ефективността спрямо броя нишки.



Използвани източници

- Манделбълб
  - http://mandelbulb.com/3d-fractal-art-mandelmorphs/
  - https://en.wikipedia.org/wiki/Mandelbulb
  - http://blog.hvidtfeldts.net/index.php/2011/09/distance-estimated-3d-fractals-v-the-mandelbulb-different-de-approximations/
  - http://blog.hvidtfeldts.net/index.php/2011/06/distance-estimated-3d-fractals-part-i/
- Cuda
  - https://en.wikipedia.org/wiki/CUDA
  - http://docs.nvidia.com/cuda/
  - http://www.hds.bme.hu/~fhegedus/C++/Professional%20CUDA%20C%20Programming.pdf
  - https://bluewaters.ncsa.illinois.edu/documents/10157/63094/NCSA02a-window-example-cpp-08-14-13.pdf