PROJECT PROPOSAL PHPC-2016

A High Performance Implementation of a Ray Marching Renderer

Principal investigator (PI)	Sebastien Speierer
Institution	EPFL-SC
Address	-
Involved researchers	-
Date of submission	May 21, 2016
Expected end of project	July 1, 2016
Target machine	Deneb1
Proposed acronym	RAYMARCHING

Abstract

Rendering algorithms are highly used in the movie industry to create computer generated images. Most of the time, 3D scenes and models are represented using sets of primitive geomtries, usually triangles. Due to the complexity of triangle intersection computations with large meshes, we propose in this project to use a less common type of algorithms called ray marching which will allow us to quickly render complex models. In our case, models are represented by distance field functions and the intersection test is done in an iterative way. 3D fractals such as the Mandelbulb and the Menger Sponge are really suitable for this kind of algorithms.

1 Scientific Background

In this section, I explain the foundations of rendering algorithms and then talk in more in detail about the ray marching algorithm.

1.1 Basic of rendering

Rendering algorithms generate images given a set of 3D models called a scene. As in the real world, we can move a virtual camera and change its direction in our 3D world. This camera is described by a few parameters such as its position, direction, field of view, ... More advanced camera features are discribed in Section 1.3.

Ray Tracer

The most common rendering algorithm is called ray tracing. It throws rays into the scene for each pixel and checks the intersection of this ray with every triangles composing the different models. This method is capable of producting really realistic images, but at a greater computational cost. More advanced implementations of ray tracer can handle a wide variety of optical effects, such as reflection, refraction, shadows, motion blur, ...

The number of triangles in a model increases proportionally to its complexity. In order to render more complex models such as the 3D Mandelbulb fractal (Figure 2b), it would become

necessary to use specialized data structures such as bounding volume hierarchy or K-d tree which are pretty hard to implement on GPUs in a parallel fashion.

However, an other algorithm, called **ray marching**, uses distance field functions to represent models in the scene instead of triangles. It allows us to quickly render really complex (infinitly complex models such as fractals) models as easily as a ray tracer would render one triangle.

Ray marching

The ray marching method is said to use an iterative intersection check since it takes several steps for a ray to reach the hit point on the surface of the primitive. With ray tracing, the intersection check tells you exaclty where a ray hits the primitive. On the other hand, with ray marching, the intersection check gives you an estimation of how close you are to the surface. Then, it uses this information to take a step futher toward the surface of the primitve until it gets as close as desired.

Here is a pseudo code for the ray marching algorithm:

Data: Camera position $\vec{c_p}$, camera direction $\vec{c_d}$, distance threshold ϵ
distance function $DE(p)$
Result: Rendered image $I(x, y)$
for $x \leftarrow 0$ **to** *SCREEN WIDTH* **do**
  for $y \leftarrow 0$ **to** *SCREEN HEIGHT* **do**
    generate a ray direction $\vec{d}$;
    $d_{current} = DE(\vec{c_p})$;
    $d_{tot} = 0$;
    while $d_{current} > \epsilon$ **do**
      $d_{current} = DE(\vec{c_p} + d_{tot} \cdot \vec{c_d})$;
      $d_{tot} = d_{tot} + d_{current}$;
    end
    $\vec{h}_{point} = \vec{c_p} + d_{tot} \cdot \vec{c_d}$;
    Compute the color of the pixel $I(x, y)$ given the hitting point $\vec{h}_{point}$
  end
end

and Figure 1 shows the iterative intersection process for ray marching method, taking several steps to eventually reach the model.

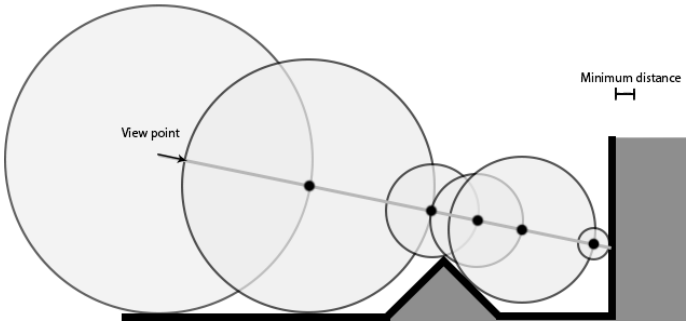


Figure 1: Ray marching iterative intersection check

1.2 Distance field function

Distance field functions are used to represent our virtual objects. They take a position as parameter and return an estimation of the distance between this point and the object. Here are a few examples of distance field functions (noticed they're all centered at the origin):

- The distance field function for a **sphere**, defined only by its radius: $f_{sphere}(\mathbf{p}) = ||\mathbf{p}|| - r$
- The distance field function for the horizontal infinite **plane**: $f_{plane}(\mathbf{p}) = p_z$

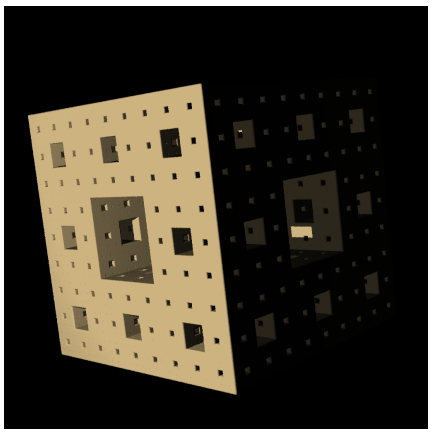
Menger Sponge

The **Menger Sponge** was first described in 1926 by Karl Menger, an Autrian mathematician. The construction of the fractal starts with a cube and recursively divide it into 27 smaller cubes. At each recursion step, remove the small cube at the center and at the center of every faces. Figure 2a shows the results of this process with 4 recursions. The source code of its distance function is given in Appendix A. It basically recursively substracts a cube and a 3D cross shape.

Mandelbulb

For several years, mathematicians tried to find a way of creating a 3D version of the well-known mandelbrot set. The **Mandelbulb** was discovered in 2009 and uses spherical coordinates but is not a real 3D extention of the mandelbrot 2D set.

The process of creating the Mandelbulb starts from extracting the polar coordinates of a 3D point, doubling its angles and raising it's length to a power N. The most famous results (and the one you can see on Figure 2b) are given using $N = 8$. There are no specific mathematical justification for this formula but the fact that is yields interesting and beautiful results. The source code of its distance function is given in Appendix A.



(a) Menger Sponge with 4 iterations

(b) Mandelbulb with $N = 8$

Transformation

It is really easy to play around with these distance field functions. With some simple tricks, it's possible to translate and scale primitives or even apply some more fancy transformations. For instance, applying the modulo operator to the result of the distance field function will infinitly repeat the object.

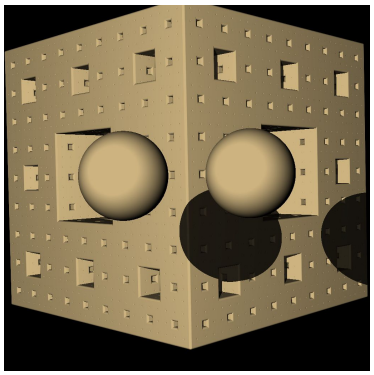
Combinations

In order to create new models, we can also combine simpler models together and get something totally new. The most common operator of this kind is the union operator. It is used to display several primitives at the same time. Mathemetically, it is implemented by the min operator:

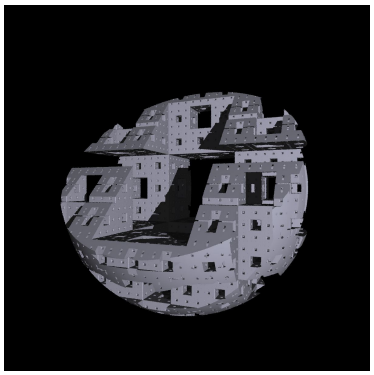
$$(DE_1 \cup DE_2)(\vec{p}) = \min(DE_1(p), DE_2(p))$$

Another useful operator is the intersection operator which is mathematically implemented by the max operator:

$$(DE_1 \cap DE_2)(\vec{p}) = \max(DE_1(p), DE_2(p))$$



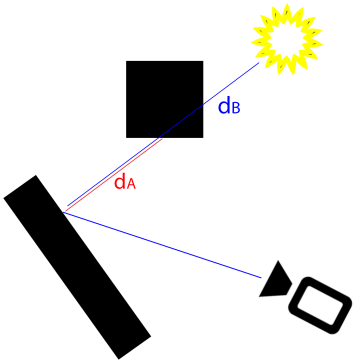
(a) Union of two spheres and a Menger Sponge



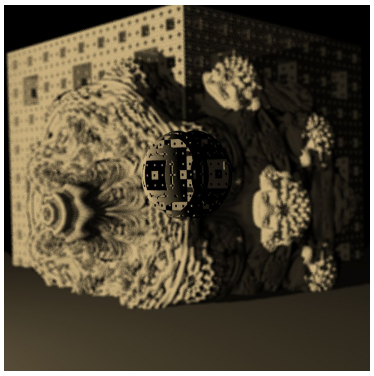
(b) Intersection of a sphere and the menger sponge

1.3 Other features

Shadows occur when a object obstructs another one from a light source. The ray marching algorithm is really suitable for shadows rendering. As described on Figure 4a, it is enough to compare distance d_A and d_B to know if a point is in shadow. In practise, the algorithm simply casts another ray from the hitting point to the light source and compares the distance.



(a) Distance checking for shadow computation



(b) Depth of field example

Depth of field refers to the range of distance that appears acceptably sharp on an image. This optical effect is really easy to simulate by simply casting many rays per pixels and sampling direction according to the optics of lenses. Like in all modern renderers, my algorithm implements this feature and Figure 4b shows a results of it.

2 Implementations

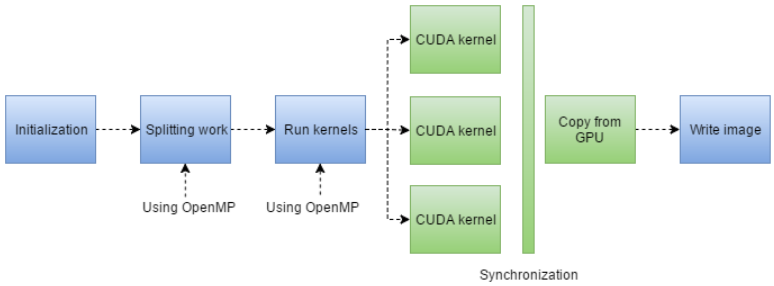
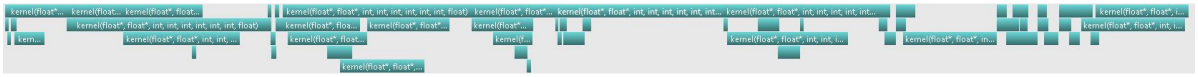


Figure 5: Overall pipeline

The ray marching algorithm is embarrassingly parallel and therefore is really suitable for an HPC project. For my implementation, I mainly use CUDA, an API providing to developers the ability to use GPUs for general purpose computation. Thanks to their enormous parallel computation power, the use of GPUs for solving this problem is really appropriate.

I quickly realized that the overall workload distrubution over the 2D domain isn't uniform. Thus, some of the blocks of threads on the GPU will finish their task really quickly while others will spend twice the time accomplishing it. To illustrate this problem, I first subdivided the domain in many sub-domains (64) and rendered each sub-domain on a different kernel. An interesting feature of CUDA is the use of CUDA streams to execute multiple kernels simultaneously to more fully take advantage of the device's multiprocessors. Then, using the *NVIDIA Visual Profiler*, I was able to gather information concerning this work load distribution by looking at the time each kernel would spend to finish its task. It is obvious on Figure 6a that some kernels are taking way longer than others.



(a) Uniform domain subdivision

(b) Smart domain subdivision

Figure 6: Kernel execution observed on NVidia Visual Profiler

Therefore, a good way to increase performance on this problem would be to distribute the work more fairly to the different kernels, which is done in my recursive load balancing algorithm.

2.1 Recursive load balancing

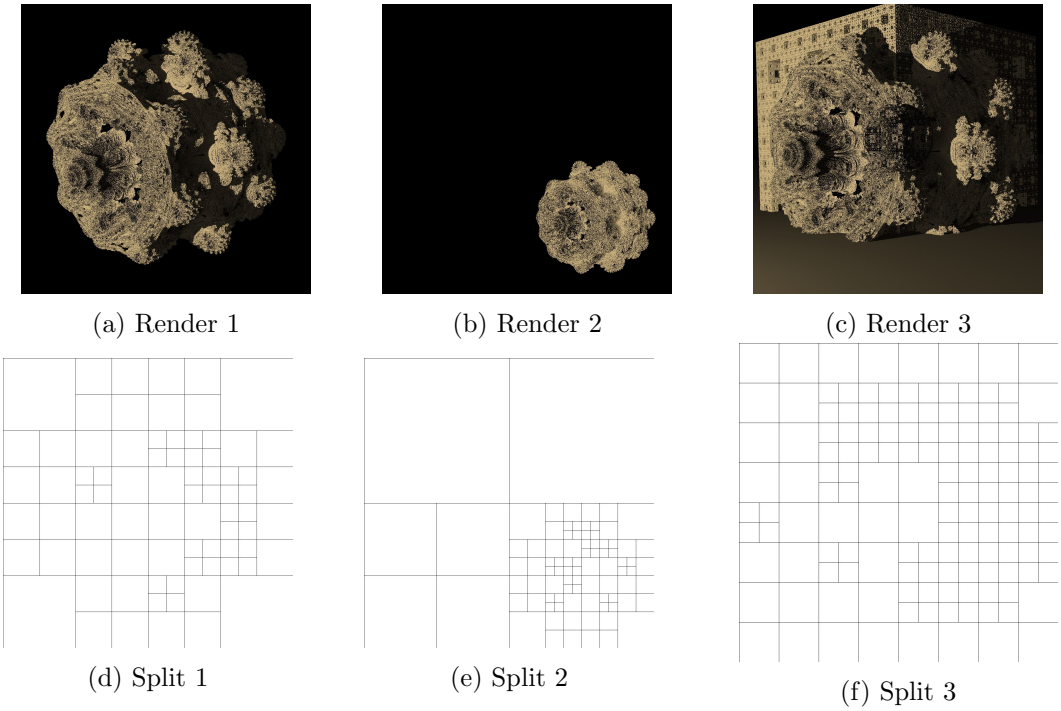
The basic idea of this recursive load balancing method is the following: *Splitting: For a given sub-domain, estimate the time needed to render this chunk. If the estimated time is greater than a threshold, divide this sub-domain into two (or four) sub-domains and recursively apply the splitting process on them.*

In this way, with a proper threshold, more complex sub-domains will be subdivided a step futher. As a result of this recursive process, computation time of the kernels rendering these

sub-domains will be more equal, as shown on Figure 6b. Figure 7 shows the sub-domain splits on different scene configurations. As you can observe, smaller sub-domains are created on the fractals since their computations are more costly.

To compute a approximation of the time needed for the rendering of a given sub-domain, we simply samples a few rays on this sub-domain and render them. By timing this operation, we can approximate the time needed for rendering the entire sub-domain, even with only a few samples. The threshold is computed at the first depth level of the recursive splitting process and is given by $\frac{\tilde{T}_0}{64}$ with $\tilde{T}_0$ the estimate duration for the rendering of the entire image. The justification of this formula is that we want to avoid having too many kernels (more than ~ 150) launching on the GPU because of the overhead of each kernel call. Also, if the workload was uniformly distributed on the image, it would give us 64 kernels which is fine. And even in the case where some of these 64 chunks needed twice as much time (even half of them), we would still only have less than one hundred kernels. After trying several thresholds, it seems this formula is a good trade off.

In the prospect of rendering a sequence of frames on our GPU, it make perfect sens to dedicate the sub-domain splitting operation to the CPU. In this way, it will be possible to let the CPU prepare the sub-domains for the next frame while the GPU is rendering the current frame asynchronously. So that we reduce the time spent on the splitting step of our pipeline, I also parallelize this process on the CPU using the OpenMP library. Moreover, in anticipation of the use of several GPUs for rendering very large fractals, the splitting step has to be done on the side CPU so it can then distribute the sub-domains to the different GPUs using message-passing libraries such as MPI. Unfortunately, this hasn't been explored in this project.



(a) Render 1 (b) Render 2 (c) Render 3

(d) Split 1 (e) Split 2 (f) Split 3

Figure 7: Few examples illustrating my load balancing method. On the first row are the rendered images and on the second row are the corresponding splits.

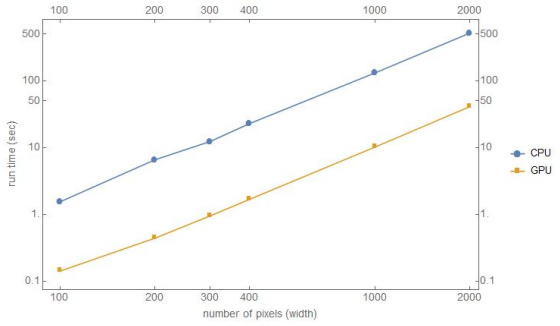
3 Results and improvement

3.1 Paralellization on the GPU

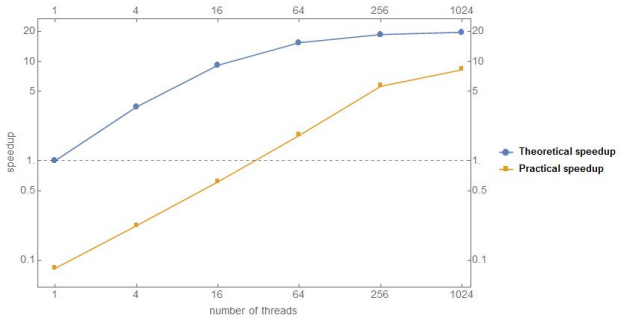
Amdahl's Law specifies the maximum speedup that can be expected by parallelizing portions of a serial program. In my ray marching implementation, most of the code has been parallelized (approximately 95%). Taking Amdahl's Law, we can then compute an upper bound of the speedup in function of the number of threads:

$$S(n) = \frac{1}{(1-P) + \frac{P}{n}} \tag{1}$$

with $P = 0.95$ the fraction of parallelized code. The GPU I used to run my implementation allows a maximum of 1024 threads per block, which should give us a theoretical speedup of ~ 20 if we use one single block. As we see on Figure 8a, the speedup given by the use of GPUs in my implementation is of the order of ~ 10.



(a) CPU performance versus GPU performance



(b) Speedup analysis

Figure 8b gives a good idea of the speedup we get using GPU on this problem. Notice that the single thread computation on the GPU is way slower than the CPU computation. This is due to the fact that CPU's cores are a lot faster than GPU's. However, as soon as we use more than ~ 50 threads on the GPU, we drastically increase the performance of our program.

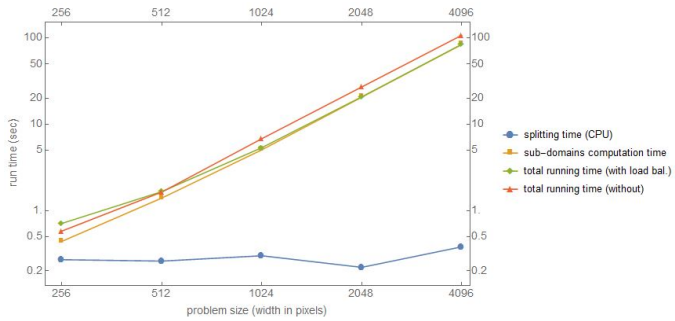
3.2 Load balancing performance

As explained in section 2.1, we recursively split our domain in sub-domains to better balance the work between threads. Figure 9a shows the results of this method. The first thing we notice is that the bigger the size of the problem is, the least we can distinguish the yellow line from the green one. It means that the splitting time (almost constant) becomes negligible when the problem becomes large enough.

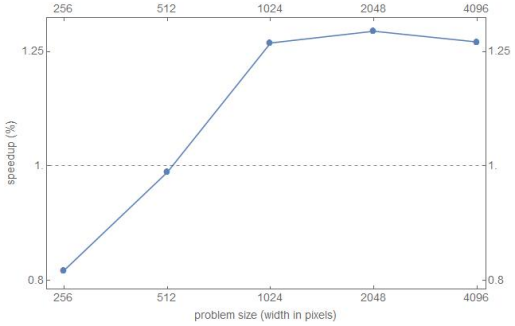
Figure 9b presents the corresponding speedup we get using the resursve load balancing method. We loose in performance when the problem is too small because at this point, splitting time isn't negligible. However, again, as soon as the problem gets larger, we gain about ~ 0.25 in performance which is a pretty good speedup.

3.3 Strong scaling

Strong scaling is a parallel performance measurement where the problem size stays fixed but the number of processing elements (here threads in GPU blocks) are increased.



(a) Load balancing comparison of the running times



(b) Recursive load balancing speedup

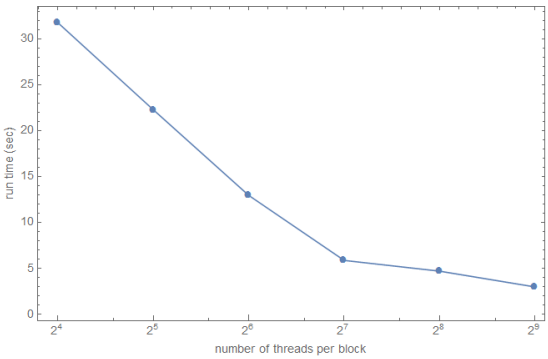
In order to measure the strong scaling, I fixed the problem size to 1024x1024 pixels and I varied the number of threads per blocks. As expected, the run time decreases while the number of thread increases as shown on Figure 10a. Another interesting measurement is the efficiency measurement of the strong scaling. It is given by the following formula:

$$e_n = 100 \cdot \frac{x_1 t_1}{x_n t_n}$$

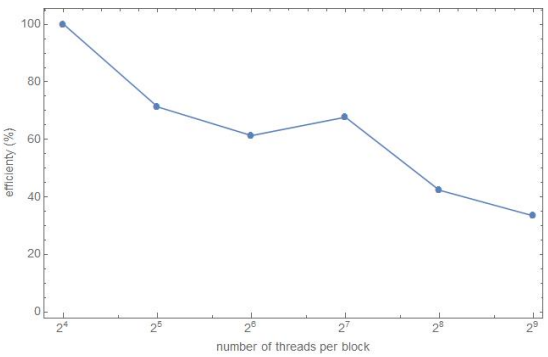
with x_i the number of threads and t_i the corresponding computation time. Figure 10b demonstrates a loss in efficiency due to parallel overhead when increasing the number of threads per block.

There exist different justifications for this loss in efficiency. Since I get similar trends with and without the recursive load balancing, I will give a justification for both cases:

- *Without*: As mentioned in the previous section, due to the unfair load balance in the work distribution among the threads, threads are sometimes idle, waiting for other to finish. This could be one of the reason for such a lack of strong scaling.
- *With*: Even with good load balancing method, the computation time of each thread varies a little bit and it prevents the algorithm to reach the theoretical speedup. Moreover, launching many kernels on the device comes with a cost that also influence the strong scaling ability of my method.



(a) Strong scaling

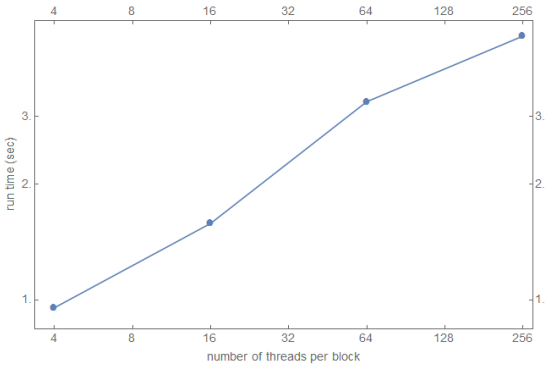


(b) Strong scaling efficiency

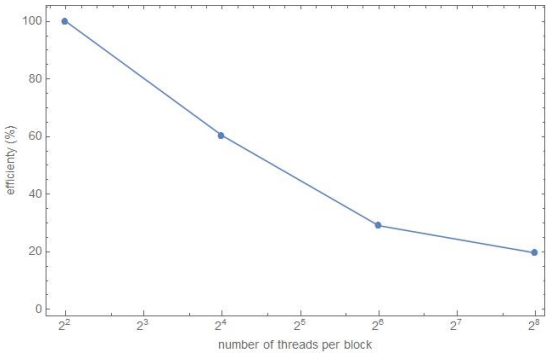
3.4 Weak scaling

In weak scaling, the problem size assigned to each processing element stays constant. In my case, I fixed the number of pixels a thread will compute to 16. For instance, for computing a 128x128 image, I will use 1024 threads or a block of 32x32 threads.

In the case of weak scaling, linear scaling is achieved if the run time stays constant while the workload is increased in direct proportion to the number of processors. Figure 11a presents the results of weak scaling on my program and it is clear that it isn't linear scaling. What it means is that the parallel overhead in my implementation are taking over as the problem size increases. It is good to notice that I get similar results without using the recursive load balancing.



(a) Weak scaling



(b) Weak scaling efficiency

Weak scaling efficiency is computed using the following formula:

$$e_n = 100 \cdot \frac{t_1}{t_n}$$

Again, due to high parallel overhead, my implementation doesn't scale really well and looses efficiency as soon as we increase the number of threads as we can see on Figure 11b.

3.5 Memory

Another system resource that can often have an impact on an HPC program efficiency is the memory. However, this is not a problem in my case. Since the virtual objects are represented using distance field functions, only the array of pixels actually consumes memory on the GPU and the host. Even for a colorful high-resolution image (2048x2048 pixels) which would take approximately 50 MBs, it is still pretty low compared to the gigabytes per second transfer bandwidth that we can achieve on today's GPUs. On top of that, we only need to transfer it once at the end of all computations for such an image.

Using the *NVIDIA Visual Profiler*, I could measure the time spent on memory transfer from the device to the host. As expected, these times are negligible. The copy time for a 1024x1024 image was approximately 1 milliseconds, which is $\sim 4.5 \cdot 10^{-4}\%$ of the overall computation time.

4 Resources budget

Looking at the specifications of my GPU, I could compute an approximation of the number of floating-point operations it would take to compute a 4k mandelbulb. My GPU has a computational capacity of 691.2 GFlops. Since it takes 8.210000 seconds to render a 2048x2048 mandelbulb, I can approximate the number of floating-point operations to $5.674 \cdot 10^{12}$ so $1.353 \cdot 10^{6}$

per pixels. 4k images have a resolution of 3840x2160 or $8.29 \cdot 10^6$ pixels which means it would take $1.1222323 \cdot 10^{13}$ floating-point operations to compute such an image.

Knowing that *DENEB* has 16 GPU nodes and that on each of these nodes runs a *NVIDIA Tesla K40*, we can estimate the computation time for a 4K fractal using only one node to 7.84778 seconds. As a comparison, it would approximately take 16.2407 on my GPU. In order to exploit the real power of the cluster, it would be necessary to implement an MPI version of my program which would take advantage of several GPU nodes in parallel. In this case, it would approximately take 0.5 seconds to render this 4K fractal.

Notice that these numbers are really sensitive to the type of fractals we render or even the view point from which we render it. Please consider these numbers as rough approximation of what they would really be in practise.

5 Scientific outcome

Fractals are beautiful! For more than 25 years now, people have been fascinated by 2D and 3D fractals representation using computers. In order to explore all the features and patterns of a fractal, high-resolution images are necessary and thus high-performance computation power.

The allocation of powerfull GPUs in your HPC clusters would give us the opportunity to explore deeper areas of the fractal world and render amazing spectacle from this infinite world. Also, in the case of frames rendering for movies, knowing that today's productions have more than 30 images per seconds, it becomes clear that performance is a real issue in this industry.

A Distance field functions source code

Mandelbulb

```
float mandelbulb(const Vector p) {
  int power = 8;
  Vector z = p;
  float dr = 1.0;
  float r = 0.0;
  int i;
  for (i = 0; i < 22 ; i++) {
    r = norm(z);
    if (r > 2.5) break;

    // convert to polar coordinates
    float theta = acosf(z.z / r);
    float phi = atan2f(z.y , z.x);
    dr = powf(r, power - 1.0) * power * dr + 1.0;

    // scale and rotate the point
    float zr = powf(r, power);
    theta = theta * power;
    phi = phi * power;

    // convert back to cartesian coordinates
    z.x = zr * sinf(theta) * cosf(phi);
    z.y = zr * sinf(phi) * sinf(theta);
    z.z = zr * cosf(theta);
    z = add(z, p);
  }
  return 0.5 * log(r) * r / dr;
}
```

Menger Sponge

```
float menger_DE(const Vector pos){
  int n; int iters = 8;
  Vector b = {1.5, 1.5, 1.5};
  float t; float x = pos.x;
  float y = pos.y; float z = pos.z;
  for(n = 0; n < iters; n++){
    x = fabsf(x);
    y = fabsf(y);
    z = fabsf(z); //fabsf is just abs for floats
    if(x < y)
      t = x; x = y; y = t;
    if(y < z)
      t = y; y = z; z = t;
    if(x < y)
      t = x; x = y; y = t;

    x = x * 3.0 - 2.0; y = y * 3.0 - 2.0; z = z * 3.0 - 2.0;

    if(z < -1.0)
       z += 2.0;
  }
  x = fmaxf(fabsf(x) - b.x , 0.0);
  y = fmaxf(fabsf(y) - b.x , 0.0);
  z = fmaxf(fabsf(z) - b.x , 0.0);
  return (sqrt(x * x + y * y + z * z) - 1.0) * powf(3.0, -(float)iters);
}
```

References

[1] Eric Haines, Naty Hoffman, and Tomas Mller, *Real-Time Rendering* Inc.,1999

[2] Dagum L. and Menon R., *OpenMP: An Industry-Standard API for Shared-Memory Programming*, IEEE Computational Science & Engineering, Volume 5 Issue 1, pp 46-55, January 1998

[3] CUDA Toolkit Documentation, `http://docs.nvidia.com/cuda/`, 2013

[4] CUDA Pro Tip: nvprof is Your Handy Universal GPU Profiler, `https://devblogs.nvidia.com/parallelforall/cuda-pro-tip-nvprof-your-handy-universal-gpu-profiler/`, 2013