A Comparison between GPU-based Volume Ray Casting Implementations: Fragment Shader, Compute Shader, OpenCL, and CUDA

Francisco Sans

Computer Graphics Center, Computer Science Department,
Science Faculty, Central University of Venezuela,
Caracas, Venezuela. 1041-A
francisco.sans@ciens.ucv.ve

and

Rhadamés Carmona

Computer Graphics Center, Computer Science Department,
Science Faculty, Central University of Venezuela,
Caracas, Venezuela. 1041-A
rhadames.carmona@ciens.ucv.ve

Abstract

Volume rendering is an important area of study in computer graphics, due to its application in areas such as medicine, physic simulations, oil and gas industries, and others. The main used method nowadays for volume rendering is ray casting. Nevertheless, there are a variety of parallel APIs that can be used to implement it. Thus, it is important to evaluate the performance of ray casting in different parallel APIs to help programmers in selecting one of them. In this paper, we present a performance comparison using OpenGL[®] with fragment shader, OpenGL[®] with compute shader, OpenCL, and CUDA.

Keywords: GPU, OpenGL, OpenCL, CUDA, Performance Comparison, Volume Ray Casting, Parallel APIs, Compute Shader.

1 Introduction

The visualization of volume datasets has gain importance in the last decades in areas such as medicine, physic simulations, oil and gas industries, and others. Several methods has been used to evaluate the volume rendering equation numerically: viewport aligned planes (using 3D texture), object aligned planes (using 2D texture), shear-warp, splatting and ray casting [1]. Ray casting is currently the standard in volume visualization, due to its parallel nature and rendering quality. Since this method is highly parallel, GPU-based ray casting is an attractive option to implement it with interactive frame rates and low cost.

One of the problems to be considered for GPU-based ray casting is the ray/volume intersection. There are two main methods to calculate this intersection: rasterization [2] and ray/box intersection [3]. With the entry and exit point of one ray into the volume, the problem is commonly reduced to sampling the volume at constant steps, classifying the samples and composing them. It is also accompanied with acceleration techniques like early ray termination [4].

There are several APIs which allow the implementation of general purpose parallel programs in the GPU. The most commonly used are OpenCL [5] and CUDA [6]. For ray casting, using OpenGL[®] [7] pipeline should be a good option, since it is optimized for graphics problems; however, not all the stages of the pipeline are programmable, and the programmable stages are mainly limited to process geometries, vertexes and fragments [7]. Nevertheless, nowadays OpenGL[®] 4.5 allows general purpose computing with compute shader [8]. One of the advantages of using compute shader over CUDA or OpenCL is that it is integrated directly into OpenGL[®], which could be significantly important for a graphic-driven application.

In this paper, we do a performance comparison of four volume ray casting implementations, using: OpenGL[®] with fragment shader, OpenGL[®] with compute shader, OpenCL, and CUDA. With this study we hope to answer two main question in the developing of a volume ray casting: (1) which of these four versions is currently the most efficient approach for ray casting in a given system? (2) which ray/volume intersection approach is better to use in terms of speed?.

The rest of this paper is organized as follows: Section 2 makes a survey of the related work. Section 3 briefly describes volume ray casting, showing how this method is optimized with parallel APIs. Then, Section 4 shows some details of the volume ray casting implementations used for testing. In Section 5, test datasets and results are presented. Finally, Section 6 discusses the conclusions and future work.

2 Related Work

GPUs have become an important tool to accelerate calculation due to their versatility and powerful parallel processors. For that reason, traditional volume rendering approaches have been adapted to exploit the use of this hardware. The earliest methods to exploit the GPU were the 2D and 3D texture slicing [9][10]. The disadvantage of using 2D texture slicing was the lack of tri-linear interpolation between the slices, the necessity to have three copies of the volume, and the bright differences in the final render when changing between copies. In the case of 3D texture slicing, there were differences between volume samples distance per ray due to the perspective projection. That was fixed with the use of spherical shells, but it adds the necessity of more complex geometries. Furthermore, these approaches often exhibit visual artifacts and less flexibility when compared to ray casting methods. For ray casting on GPUs, Roettger et al. [11] and Kruger and Westermann [2] were among the first to introduce this method, using a multi-pass approach. Two years later, Hadwiger et al. [12] implemented a single pass ray casting.

To know which volume rendering method is better, the performance of different methods has been studied; Lacroute [13] compared shear-warp with ray casting and splatting. Nevertheless, our work focuses directly in the performance comparison of volume ray casting, aiming the use of OpenGL[®], OpenCL, and CUDA. Even though those APIs have similar capabilities, they present some differences that could make one API be a better option than others under certain circumstances. Different authors have evaluated the performance of those APIs individually. Garland et al. [14] made a survey of experiences gained in applying CUDA to a diverse set of problems and reported the parallel speedups over sequential codes running on CPU. Shen et al. [15] analyzed the performance of OpenCL in multi-core CPUs, and compare its performance with OpenMP.

Moreover, other authors have compared the performance between two or more GPU parallel APIs. OpenCL and CUDA were compared in different studies with SystemC simulation [16], Monte Carlo simulation of a quantum spin system [17], and several other programs [18][19]. In general, those works showed that CUDA runs faster than OpenCL in the default mode, and with some optimizations, the performance of OpenCL is comparable with CUDA. Satish et al. [20] implemented a CUDA sorting algorithm which demonstrated to be 4 times faster than a GPU sorting algorithm using the OpenGL[®] pipeline. OpenGL[®], OpenCL, and CUDA were also compared with implementations of the cardiac monodomain equations [21] and cloth modelling [22]. Those studies showed that there is not a unique best API choice for all problems, since the best API choice varies from problem to problem.

Some studies have focused in measuring the performance of volume ray casting using a parallel API. Marsalek et al. [23] demonstrated proof of concept of a simple CUDA ray caster. Schubert and Scholl [24] have implemented a multi-volume renderer using CUDA with different mixing techniques, obtaining real time results. The performance increase of a CUDA based volume rendering with respect to a CPU have also been measured in [25] and [26]. Fangerau and Kromer [25] claim that their implementation runs faster with CUDA using a block size configuration of 8×8 .

Furthermore, other authors have compared the performance of volume ray casting between different implementations. On one hand, Kiss et al. [27] compared a CUDA and OpenCL volume rendering implementation, obtaining similar milliseconds per frame with both technologies. On the other hand, other comparisons between the OpenGL[®] fragment shader and CUDA have been made. Weinlich et al. [28] did a comparison with OpenGL[®] and CUDA with compute capability 1.1 and 2.0. They concluded that an implementation with compute capability 1.1 is too slow due to the lack of 3D texture, and that an implementation with compute capability 2.0 is slightly better than an OpenGL[®] implementation. Shi et al. [29] implemented a volume ray casting with OpenGL[®] and CUDA, showing an improved performance of CUDA by about 15%, but they used small datasets in their test. Mensmann et al. [30] took advantage of the CUDA capabilities to implement an OpenGL[®] and CUDA slab-based ray casting for bricked volumes. The results showed that CUDA is slightly better than OpenGL[®] for their implementation. In general, CUDA showed a better performance than the use of OpenGL[®] or OpenCL.

In our work, we make a performance comparison of volume ray casting using fragment shader, compute shader, OpenCL and CUDA. Unlike previous work, we compare more than two ray casting parallel implementations and we take into account the use of compute shader. This shader was not tested for ray casting in previous work to the best of our knowledge.

3 Volume Ray Casting

Volume rendering is a technique for rendering a three dimensional scalar field by projecting its samples without requiring an intermediate reconstruction of the data [31]. This technique simulates the interaction between the light and a participating media (the volume), and is modeled with the Equation 1.

$$C = \int_0^D c(x(\lambda))\tau(x(\lambda))e^{\int_0^\lambda \tau(x(\lambda'))d\lambda'} d\lambda \quad (1)$$

The values $c(x(\lambda))$ and $\tau(x(\lambda))$ represent the color and absorption of a sample at a distance λ , and $x(\lambda)$ represents the parametrization of the ray [32]. There are several ways to evaluate the rendering equation. Generally, the ray is quantized in equally-spaced samples, and a transfer function is applied to each sample to perform a composite operation between classified samples. Ray casting evaluates directly the volume rendering equation by casting a ray for every pixel in the final image, with the origin in the position of the eye as can be depicted in Figure 1.

Figure 1: Calculate the color of an image pixel from the contributions of color and opacity of the equidistant ray samples

The volume rendering equation can be approximately discretized as shown in Equation 2, where α_i and c_i are the opacity and the color of the i -th sample respectively.

$$C \approx \sum_{i=0}^{n-1} \prod_{j=0}^{i-1} (1 - \alpha_j) \alpha_i c_i \quad (2)$$

As volume ray casting evaluates the volume rendering equation directly, it commonly achieves better visual results than other approaches. Furthermore, as the calculation is independent for every ray, the process can be easily parallelized.

To start with the traversal of the ray, the entry and exit point of the ray in the volume need to be calculated. The way in which this intersection is calculated could represent an important impact to the overall performance of the algorithm. This intersection is generally calculated with two approaches: (1) rasterization [2], and (2) ray/box intersection test [3].

3.1 Intersection using Rasterization

One efficient way to obtain the intersection between view-rays and the volume is using OpenGL[®] rasterization. The idea is to use the parallel rasterization power of the graphics cards to generate the volume/ray intersection for each pixel of the framebuffer. A simple color cube, representing the boundaries of the volume, is rendered (see Figure 2) [2]. Since the colors and the texture coordinates are represented in unit space $[0, 1]^3$, each color of the cube also represents a 3D texture coordinate of the volume. Thus, the color of each rasterized pixel represents the texture coordinate of the intersection of one ray with the volume. This method is used with back face culling to obtain the first hit of each ray (entry point, Figure 2a), and with

front face culling to obtain the last hit (exit point, Figure 2b). Furthermore, the direction of the ray can be obtained as exit point minus entry point (Figure 2c).

Figure 2: Images obtained by rasterization that contain: (a) the first hit of the ray with the volume, (b) the last hit of the ray with the volume, and (c) the direction of the ray, which can be obtained with the difference of the color intensity of image (b) and (a)

3.2 Ray/Box Intersection Test

The intersection of one individual ray with the volume, can also be obtained directly by using some mathematics. A divide and conquer algorithm is commonly used. Intersecting a ray with a box can be reduced to intersecting the ray with each slab of the volume [3]. This test has been optimized to work in parallel architectures, avoiding thread divergence [33]. An implementation of this test can be depicted in Listing 1.

```
bool intersectBox(Ray r, out float tnear, out float tfar)
{
    //unitary volume bounding box
    vec3 boxmin = vec3(-.5f, -.5f, -.5f);
    vec3 boxmax = vec3(.5f, .5f, .5f);

    //compute intersection of ray with all six box planes
    vec3 invR = 1.0f / r.d;
    vec3 tbot = invR * (boxmin - r.o);
    vec3 ttop = invR * (boxmax - r.o);

    //re-order to find smallest and largest value on each axis
    vec3 tmin = vec3(100000.0f);
    vec3 tmax = vec3(0.0f);

    tmin = min(tmin, min(ttop, tbot));
    tmax = max(tmax, max(ttop, tbot));

    //find the largest tmin and the smallest tmax
    tnear = max(max(tmin.x, tmin.y), tmin.z);
    tfar = min(min(tmax.x, tmax.y), tmax.z);

    return tfar > tnear;
}
```

Listing 1: General code for a ray/box intersection test optimized to avoid thread divergence

4 Implementations

The implementation of volume ray casting in different parallel APIs have the same basic algorithm. Nevertheless, every API has its own limitations that is worth mentioning. This Section presents some implementation details of the volume ray casting used in this work, considering fragment shader, compute shader, OpenCL, and CUDA. We only show the implementations with the rasterization method, as the ray/box intersection algorithm was previously presented in Listing 1, and that algorithm could be easily adapted to each API. Furthermore, this Section presents an implementation for adding diffuse lighting to the volume ray casting.

4.1 Fragment shader

Given that it is necessary to render some geometry with OpenGL[®] to activate a fragment shader program, we can compute the intersection of every ray with the volume during rasterization, with minimum extra overhead. In this case, it will only be necessary to do two rendering passes. The first pass calculates the exit point of the ray. The second pass calculates the entry point of the ray, the ray direction, and numerically evaluates the volume rendering equation, as can be depicted in Listing 2.

```
layout(binding = 1) uniform sampler2D lastHit;
layout(binding = 2) uniform sampler1D transferFunction;
layout(binding = 3) uniform sampler3D volume;
uniform float h;

in vec3 first;

#define opacityThreshold 0.99

layout(location = 0) out vec4 vFragColor;

void main(void){
    ivec2 tcoord = ivec2(gl_FragCoord.xy);
    vec3 last = texelFetch(lastHit, tcoord, 0).xyz;

    //get direction of the ray
    vec3 direction = last.xyz - first.xyz;
    float D = length(direction);
    direction = normalize(direction);

    vec4 color = vec4(0.0f);
    color.a = 1.0f;

    vec3 tr = first;
    vec3 rayStep = direction * h;

    for(float t = 0; t <= D; t += h){
        //sample the scalar field and the transfer function
        vec4 samp = texture(transferFunction,
                           texture(volume, tr).x).rgba;

        //calculating alpha
        samp.a = 1.0f - exp(-0.5f * samp.a);

        //accumulating color and alpha using under operator
        samp.rgb = samp.rgb * samp.a;

        color.rgb += samp.rgb * color.a;
        color.a *= 1.0f - samp.w;

        //checking early ray termination
        if(1.0f - color.w > opacityThreshold) break;

        //increment ray step
        tr += rayStep;
    }

    color.w = 1.0f - color.w;
    vFragColor = color; //Set the final color
}
```

Listing 2: Volume rendering implemented with the fragment shader

During the second stage, the front faces of the volume cube are rendered. For each rasterized pixel, we get the entry point of the ray into the volume. It is represented by the interpolated variable **first**. Required textures are indicated to the shader through **samplers**; they represent the exit point of the ray per pixel, the transfer function, and the volume itself. The exit point of the ray (**lastHit**) is fetched with the function **texelFetch**. With **first** and **lastHit**, it is possible to calculate the direction (**direction**) and the length (**D**) of the ray. Then, the volume is transversed by the ray, step by step, in a loop. On each step, the volume is sampled at the current ray position, and the position is moved **h** units in the ray direction. For accessing the textures of the transfer function and the volume, we use the function **texture** as we want to obtain interpolated texels and voxels respectively. The code is optimized with an early ray termination; it means, when the ray accumulates an opacity value greater than a threshold (**opacityThreshold**) the loop finishes. Then, the resulting color and opacity is stored in the currently attached framebuffer, which is indicated with the output variable **vFragColor**.

4.2 Compute shader

Listing 3 shows the implementation of the volume ray casting using compute shader. The logic and the code is similar to the one presented for the fragment shader. In this case, the rasterized images with the entry and exit point of the ray are rendered with OpenGL[®] and stored in an OpenGL[®] texture. Compute shader allows the access of texture in the same way as in the fragment shader using `samplers`. The only difference is the output texture, that has to be linked as an image with `glBindImageTexture` function.

```
//Store into a texture
layout(binding = 0, rgba8) uniform writeonly
    image2D destTex;
layout(binding = 1) uniform sampler1D transferFunc;
layout(binding = 2) uniform sampler3D volume;
layout(binding = 3) uniform sampler2D lastHit;
layout(binding = 4) uniform sampler2D firstHit;

uniform float h, width, height, depth;
#define opacityThreshold 0.99

void main(){
    //read current global position for this thread
    ivec2 storePos = ivec2(gl_GlobalInvocationID.xy);

    //calculate the global number of threads
    ivec2 size = imageSize(destTex);
    if(storePos.x < size.x && storePos.y < size.y){
        storePos.y = size.y - storePos.y;
        vec3 last = texelFetch(lastHit, storePos, 0).xyz;
        vec3 first = texelFetch(firstHit, storePos, 0).xyz;

        //get direction of the ray
        vec3 direction = last.xyz - first.xyz;
        float D = length(direction);
        direction = normalize(direction);

        vec4 color = vec4(0.0f);
        color.a = 1.0f;

        vec3 tr = first;
        vec3 rayStep = direction * h;

        for(float t = 0; t <= D; t += h){
            //sampling scalar field and transfer function
            float scalar = texture(volume, tr).x;
            vec4 samp = texture(transferFunc, scalar).rgba;

            //calculating alpha
            samp.a = 1.0f - exp(-0.5 * samp.a);

            //accumulating color and alpha using under operator
            samp.rgb = samp.rgb * samp.a;

            color.rgb += samp.rgb * color.a;
            color.a *= 1.0f - samp.w;

            //checking early ray termination
            if(1.0f - color.w > opacityThreshold) break;

            //increment ray step
            tr += rayStep;
        }

        color.w = 1.0f - color.w;
        imageStore(destTex, storePos, color);
    }
}
```

Listing 3: Volume rendering implemented with compute shader

To launch the parallel kernel, the function `glDispatchCompute` must be called, indicating the total number of threads and the number of threads per block. Unlike the fragment shader, a kernel is executed for every thread and the variable `gl_GlobalInvocationID` differentiates one thread from another. As the execution of the kernel is asynchronous, the function `glMemoryBarrier` must be called with the parameter `GL_SHADER_IMAGE_ACCESS_BARRIER_BIT` to ensure that the result is written into the texture before it is used in the final rendering. Finally, a full screen quad is rasterized with the output texture into the screen.

4.3 OpenCL

OpenCL implementation of the volume ray casting is presented in Listing 4. In this case, a different API than OpenGL[®] is used to calculate the volume rendering image result, and some interoperability functions

must be used to allow the communication of both APIs. First, all OpenGL[®] textures must be bound to an OpenCL image using `clCreateFromGLTexture`. Before invoking the kernel, OpenCL must indicate to OpenGL[®] every texture to be used with `clEnqueueAcquireGLObjects`, and release them after use with `clEnqueueReleaseGLObjects`. The function `clEnqueueNDRangeKernel` must be called to launch the kernel, indicating the total number of threads and the number of threads per block.

```
#define opacityThreshold 0.99

__kernel void volumeRendering(__write_only image2d_t d_output, float constantH, unsigned int W, ←
    unsigned int H, __read_only image3d_t volume, __read_only image2d_t transferFunc, sampler_t ←
    volumeSampler, sampler_t transferFuncSampler, __read_only image2d_t firstTex, __read_only ←
    image2d_t lastTex, sampler_t hitSampler)
{
    int2 gid = (int2)(get_global_id(0), get_global_id(1));

    if (gid.x < W && gid.y < H){
        float2 Pos = (float2)((gid.x + 0.5f) / (float)W,
                               (gid.y + 0.5f) / (float)H);

        float4 color; //bg color here

        float4 aux4 = read_imagef(firstTex, hitSampler, Pos);
        float3 first = (float3)(aux4.x, aux4.y, aux4.z);
        aux4 = read_imagef(lastTex, hitSampler, Pos);
        float3 last = (float3)(aux4.x, aux4.y, aux4.z);

        //get direction of the ray
        float3 direction = last - first;
        float D = length(direction);
        direction = normalize(direction);

        color = (float4)(0.0f);
        color.w = 1.0f;

        float3 tr = first;
        float3 rayStep = direction * constantH;

        for (float t = 0; t <= D; t += constantH){
            //sample scalar field and transfer function
            float4 scalar = read_imagef(volume, volumeSampler, (float4)(tr.x, tr.y, tr.z, 0.0f));
            float4 samp = read_imagef(transferFunc, transferFuncSampler, (float2)(scalar.x, 0.5f));

            //calculating alpa
            samp.w = 1.0f - exp(-0.5 * samp.w);

            //accumulating color and alpha using under operator
            samp.x = samp.x * samp.w;
            samp.y = samp.y * samp.w;
            samp.z = samp.z * samp.w;

            color.x += samp.x * color.w;
            color.y += samp.y * color.w;
            color.z += samp.z * color.w;
            color.w *= 1.0f - samp.w;

            //checking early ray termination
            if (1.0f - color.w > opacityThreshold) break;

            //increment ray step
            tr += rayStep;
        }

        color.w = 1.0f - color.w;

        write_imagef(d_output, gid, color);
    }
}
```

Listing 4: Volume rendering implemented with OpenCL

As in the compute shader case, the function is executed once per thread and each thread is uniquely identified with the function `get_global_id`. Finally, `clFinish` must be called to ensure that the output image is ready to be rendered by OpenGL[®]. Some data type manipulation functions were added to OpenCL to make this code more readable.

4.4 CUDA

Finally, CUDA-based implementation is observed in Listing 5. As OpenCL, CUDA has functions to allow the interoperability with OpenGL[®].

```

typedef unsigned char VolType;
#define opacityThreshold 0.99

texture<VolType, 3, cudaReadModeNormalizedFloat> volume;
texture<float4, 2, cudaReadModeElementType> transFunc;
texture<float4, 2, cudaReadModeElementType> texFirst;
texture<float4, 2, cudaReadModeElementType> texLast;
surface<void, cudaSurfaceType2D> surf;

__constant__ unsigned int constantWidth, constantHeight;
__constant__ float constantH;

__global__ void volumeRenderingKernel(){
    unsigned int x = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int y = blockIdx.y * blockDim.y + threadIdx.y;

    if (x < constantWidth && y < constantHeight){
        float2 Pos = mfloat2(x, y);

        float4 first = tex2D(texFirst, x, y);
        float4 last = tex2D(texLast, x, y);

        //get direction of the ray
        float3 direction = mfloat3(last) - mfloat3(first);
        float3 tr = mfloat3(first);

        float D = length(direction);
        direction = normalize(direction);

        float4 color = mfloat4(0.0f);
        color.w = 1.0f;

        float3 rayStep = direction * constantH;

        for (float t = 0; t <= D; t += constantH){
            //sampling scalar field and transfer function
            float scalar = tex3D(volume, tr.x, tr.y, tr.z);
            float4 samp = tex2D(transFunc, scalar, 0.5f);

            //calculating alpha
            samp.w = 1.0f - expf(-0.5f * samp.w);

            //accumulating color and alpha using under operator
            samp.x = samp.x * samp.w;
            samp.y = samp.y * samp.w;
            samp.z = samp.z * samp.w;

            color.x += samp.x * color.w;
            color.y += samp.y * color.w;
            color.z += samp.z * color.w;
            color.w *= 1.0f - samp.w;

            //checking early ray termination
            if (1.0f - color.w > opacityThreshold) break;

            //increment ray step
            tr += rayStep;
        }

        color.w = 1.0f - color.w;

        //write to the texture
        uchar4 ucolor = uchar4(color.x * 255, color.y * 255, color.z * 255, color.w * 255);
        surf2Dwrite(ucolor, surf, x * sizeof(uchar4), y, cudaBoundaryModeClamp);
    }
}

```

Listing 5: Volume rendering implemented with CUDA

Almost all OpenGL[®] textures must be mapped to a CUDA array with the functions `cudaGraphicsGLRegisterImage` and `cudaGraphicsSubResourceGetMappedArray`. Then the array must be mapped to a CUDA texture with the function `cudaBindTextureToArray`. The only exception to this rule is the output image. CUDA cannot write to a texture, so the array corresponding to the output texture must be mapped to a CUDA surface with `cudaBindSurfaceToArray`. This mapping between an OpenGL[®] and a CUDA texture has only to be performed one time, and then OpenGL[®] texture will have the same GPU memory address of the CUDA texture or surface. Thus, CUDA can access the OpenGL output texture without further extra overhead.

As in the compute shader and OpenCL case, the function is executed once for each thread and a thread is uniquely identified with the variables `blockIdx`, `blockDim`, and `threadIdx`. When the kernel is launched, the number of threads per blocks and the number of blocks must be indicated. Finally, `cudaDeviceSynchronize` must be called to wait until the end of the kernel execution. Some data type manipulation functions

were added to CUDA to make this code more readable.

4.5 Diffuse Lighting

To illuminate a volume, a normal has to be obtained for every voxel using finite difference. The finite difference in a discrete scalar field as the volume, is calculated as the subtraction of the current voxel with its neighbor in every axis, which gives the gradient vector. This vector can be normalized and used as the normal vector of the volume at the corresponding world position of the voxel. The Listing 6 shows a GLSL code with a function to calculate the diffuse lighting. The same logic can be implemented easily in the compute shader, OpenCL, and CUDA.

```
uniform vec3 lightDir; //light direction in eye space
uniform vec3 diffColor; //diffuse color of the light
uniform vec3 voxelJump; //distance between voxels

//calculate lighting for the scalar sampled
vec4 litSample(vec4 samp, vec3 tr, float scalar){
    //sample neighbors
    vec3 normal;
    vec3 displacement = vec3(voxelJump.x,0.0f,0.0f); //Sample right
    normal.x = texture(volume, tr + displacement).x - scalar;
    displacement = vec3(0.0f,voxelJump.y,0.0f); //Sample up
    normal.y = texture(volume, tr + displacement).x - scalar;
    displacement = vec3(0.0f,0.0f,-voxelJump.z); //Sample back
    normal.z = texture(volume, tr + displacement).x - scalar;

    //normalize normal
    normal = normalize(normal);

    //dot product with light
    float d = max(dot(lightDir, normal), 0.0f);

    //calculate contribution
    samp.xyz = samp.xyz * d * diffColor;
    return samp;
}
```

Listing 6: Code to calculate the diffuse illumination of a voxel

5 Test and Results

In this section we show the results obtained by this research. First, the system configuration and the different datasets used for testing are presented. Then, the methodology used for every test is explained. Finally, the results are shown, alongside with the corresponding discussion.

5.1 Environment and Datasets

Test were performed in a conventional PC with Intel(R) Core(TM) i7-3770 CPU of 3.40GHz, 8 GB of RAM, Windows 7 64 bits, and a Nvidia GeForce GTX 660 with a Kepler microarchitecture. That video card supports OpenGL® 4.5, OpenCL 1.2, and CUDA with compute capability 3.0.

All tested datasets were rendered at a gazing angle with the same rotation. We selected three datasets: visible male CT from the Visible Human project [34] (Figure 3), an explicit equation (Figure 4), and a Stag Beetle [35] (Figure 5). To test different volume resolutions, the datasets were downsampled in 5 resolutions as shown in Table 1.

Furthermore, two transfer functions were used to display the male dataset. The objective of the first transfer function (a) is to define the skin as an opaque surface (male TF1). Thus, most of the rays will stop after hitting the body skin, and ray casting should finish faster. For the second transfer function (b), a semi-transparent low-density tissue and opaque bones is defined (male TF2). Thus, the rays traverse through the semi-transparent area until a full opaque voxel is reached. For the explicit equation we chose two transfer functions: (a) full opacity at first hit of some particular surface (lemn TF1) and (b) many semi-transparent surfaces (lemn TF2). Finally, the beetle dataset was rendered with two transfer functions: (a) opaque surface at the beetle exoskeleton (beetle TF1) and semi-transparent exoskeleton (beetle TF2).

5.2 Test Methodology

Test were done to compare the performance in the most fair way. First, the compute shader, OpenCL, and CUDA versions were tested with different block sizes. The block sizes were chosen to maintain a power of two

Table 1: Size of the datasets

		Male	Lemniscate	Beetle
1	Dimension	$512 \times 512 \times 1245$	$512 \times 512 \times 1058$	$832 \times 832 \times 494$
	Size(MB)	622	529	652
2	Dimension	$512 \times 256 \times 622$	$512 \times 256 \times 529$	$416 \times 416 \times 494$
	Size(MB)	155	132	163
3	Dimension	$256 \times 256 \times 311$	$256 \times 256 \times 264$	$416 \times 208 \times 247$
	Size(MB)	38	33	40
4	Dimension	$128 \times 128 \times 311$	$128 \times 128 \times 264$	$208 \times 208 \times 123$
	Size(MB)	9	8	10
5	Dimension	$128 \times 64 \times 155$	$128 \times 64 \times 132$	$104 \times 104 \times 123$
	Size(MB)	2	2	2

(a) Male TF1

(b) Male TF2

Figure 3: Rendering of male dataset with 2 different transfer functions

number of threads: 4, 8, 16, 32, 64, 128; and for each size, different configurations were tested (see Table 2). All of these configurations are considered to be tested with different width/height ratio combinations. Other block sizes and configurations were also tested (e.g. 16×16 and 32×16) in a less exhaustive way, as those configurations did not present better results than the obtained with those shown in the table. Due to lack of space, only the best result for a block size configuration are shown in the result section, for each dataset and transfer function.

Table 2: Block size configuration tested

Number of threads per block	Configurations
4	4×1 , 2×2
8	8×1 , 4×2
16	16×1 , 8×2 , 4×4
32	32×1 , 16×2 , 8×4
64	64×1 , 32×2 , 16×4 , 8×8
128	128×1 , 64×2 , 32×4 , 16×8

Furthermore, tests were performed using rasterization, and ray/box intersection test, for all datasets, all transfer functions, and with a fixed screen resolution of 1024×768 . Each pixel of the screen is mapped into a thread, and the block size indicates how those threads are grouped. If, for example, we have a 4×4 block size, then the threads will be grouped in a matrix of $1024/4 \times 768/4 = 256 \times 192$ blocks, with 16 threads per each block. The results are also compared with the ones of the fragment shader. With fragment shader there is no need of choosing a block size.

Finally, an additional test was performed adding diffuse illumination in the volume ray casting. The illumination was only tested with opaque transfer functions (as seen in Figure 6), because for transparent functions the rendering yields incoherent visual effects. This test was done for the fragment shader, compute

Figure 4: Rendering of lemniscate dataset with 2 different transfer functions

Figure 5: Rendering of Stag Beetle dataset with 2 different transfer functions

shader, OpenCL, and CUDA. To calculate the illumination, it is necessary to fetch data from neighbor voxels and to do some extra calculation in the rendering, that could end up in some performance overhead, which may differ depending on the parallel API used.

5.3 Results

First, the results of the execution time in milliseconds per frame is presented in Table 3 and Figure 7 for the fragment shader, compute shader, OpenCL, and CUDA. Those results are shown in milliseconds per frame using two methods for ray-volume intersection test: rasterization (R), and ray/box intersection test (RB). Due to lack of space, only the time for the best block size configuration is presented alongside with the corresponding block size configuration for compute shader, OpenCL, and CUDA. In the Appendix, an example table can be depicted with all block configurations for a unique dataset. For measuring R and RB tests, a timer is set at the beginning of the main loop and stops at the end, averaging the result with the number of rendered frames. We also performed some tests with both methods (R and RB), including and excluding the display of the final image into the screen, to measure the display overhead. This overhead is bounded between 0.1 milliseconds o 0.5 milliseconds in the execution time for all cases.

The results show that for larger datasets, the execution time of the volume rendering is higher, as the rays have to transverse larger volume areas. Also, the results are consistent with the transfer function transparency. The more transparent transfer function, the higher execution time, as the rays have to traverse larger volume areas.

In the table, some values are highlighted to indicate the best result between R and RB, for each imple-

Figure 6: Rendering of test datasets using diffuse illumination with a white light source

Figure 7: Comparison of the execution time of fragment shader, compute shader, OpenCL, and CUDA

Table 3: Performance comparison of fragment shader, compute shader, OpenCL, and CUDA

		Fragment		Compute R	Compute RB	OpenCL R	OpenCL RB	CUDA R	CUDA RB
male	512 × 512 × 1245	TF1	Time	68.61	37.91	36.99	60.27	60.25	47.91
			Size		4 × 4	4 × 4	8 × 8	8 × 8	4 × 4
		TF2	Time	115.77	68.59	68.20	108.57	110.97	84.24
male	512 × 256 × 622		Size		4 × 4	4 × 4	8 × 8	8 × 8	4 × 4
		TF1	Time	16.79	11.75	11.58	26.41	25.21	13.68
			Size		8 × 4	8 × 4	8 × 8	16 × 8	8 × 8
male	256 × 256 × 311	TF2	Time	31.33	21.11	20.94	44.43	43.62	26.08
			Size		8 × 4	8 × 4	8 × 8	8 × 8	8 × 4
		TF1	Time	4.00	3.86	3.85	15.11	14.98	3.90
male	128 × 128 × 311		Size		8 × 8	8 × 8	8 × 8	16 × 8	16 × 8
		TF2	Time	7.55	6.44	6.33	23.21	22.81	8.28
			Size		8 × 8	8 × 8	16 × 8	16 × 8	8 × 8
male	128 × 64 × 155	TF1	Time	2.46	2.27	2.14	11.83	11.40	2.48
			Size		16 × 8	16 × 8	16 × 8	16 × 8	16 × 8
		TF2	Time	3.46	3.88	3.78	17.84	17.51	4.13
male	128 × 64 × 155		Size		16 × 8	16 × 8	16 × 8	16 × 8	16 × 8
		TF1	Time	1.34	1.40	1.19	7.65	7.71	1.62
			Size		32 × 4	16 × 8	8 × 8	16 × 8	16 × 8
lemni	512 × 512 × 1058	TF2	Time	1.81	1.79	1.68	11.43	11.07	2.22
			Size		16 × 8	16 × 8	16 × 8	16 × 8	16 × 8
		TF1	Time	51.07	33.60	33.57	51.23	50.77	43.46
lemni	512 × 256 × 529		Size		4 × 4	4 × 4	8 × 8	8 × 8	8 × 4
		TF2	Time	110.86	72.19	71.70	109.61	110.03	91.06
			Size		4 × 4	4 × 4	8 × 8	8 × 8	4 × 4
lemni	256 × 256 × 264	TF1	Time	11.02	9.21	9.22	23.85	23.49	9.93
			Size		8 × 8	8 × 8	16 × 8	16 × 8	8 × 8
		TF2	Time	26.63	21.10	20.87	44.51	43.65	21.98
lemni	128 × 128 × 264		Size		8 × 4	8 × 4	16 × 8	16 × 8	8 × 8
		TF1	Time	3.84	3.53	3.42	14.79	14.90	3.45
			Size		16 × 8	16 × 8	16 × 8	16 × 8	16 × 8
lemni	128 × 64 × 132	TF2	Time	6.49	6.68	6.57	25.80	25.52	7.53
			Size		8 × 8	8 × 8	16 × 8	16 × 8	16 × 8
		TF1	Time	2.57	2.10	1.93	11.29	11.15	2.38
beetle	832 × 832 × 494		Size		16 × 8	16 × 8	16 × 8	8 × 8	16 × 8
		TF2	Time	3.53	3.50	3.39	18.62	18.46	3.91
			Size		16 × 8	16 × 8	16 × 8	16 × 8	16 × 8
beetle	416 × 416 × 494	TF1	Time	1.46	1.33	1.24	8.17	8.00	1.69
			Size		16 × 8	16 × 8	16 × 4	16 × 4	16 × 8
		TF2	Time	1.941	1.85	1.75	12.02	12.65	2.40
beetle	416 × 208 × 247		Size		16 × 8	16 × 8	16 × 8	8 × 8	16 × 8
		TF1	Time	104.93	69.34	69.45	144.01	138.75	84.58
			Size		8 × 4	8 × 4	8 × 8	16 × 8	8 × 8
beetle	208 × 208 × 123	TF2	Time	96.84	64.02	64.01	132.06	128.63	78.49
			Size		8 × 4	8 × 4	8 × 8	16 × 8	8 × 4
		TF1	Time	37.60	29.33	29.33	80.10	79.70	32.52
beetle	104 × 104 × 123		Size		8 × 8	8 × 8	16 × 8	16 × 8	8 × 8
		TF2	Time	36.01	27.63	27.61	73.83	73.48	30.02
			Size		8 × 8	8 × 8	16 × 8	16 × 8	8 × 8
beetle	104 × 104 × 123	TF1	Time	10.38	10.22	9.96	54.74	54.61	11.36
			Size		16 × 8	16 × 8	16 × 8	16 × 8	16 × 8
		TF2	Time	9.81	9.36	9.23	50.20	50.20	10.33
beetle	104 × 104 × 123		Size		16 × 8	16 × 8	8 × 8	8 × 8	16 × 8
		TF1	Time	5.50	5.18	5.06	33.65	33.54	6.43
			Size		16 × 8	16 × 8	8 × 8	8 × 8	16 × 8
beetle	104 × 104 × 123	TF2	Time	5.17	4.80	4.68	30.98	30.70	5.91
			Size		16 × 8	16 × 8	8 × 8	8 × 8	16 × 8
		TF1	Time	3.24	3.13	2.92	20.89	20.74	3.89
beetle	104 × 104 × 123		Size		16 × 8	16 × 8	8 × 8	8 × 8	16 × 8
		TF2	Time	3.06	2.88	2.69	19.24	19.10	3.61
			Size		16 × 8	16 × 8	8 × 8	8 × 8	16 × 8

mentation, dataset, and transfer function. In the compute shader case, RB is faster than R in 96% of the cases in up to 0.92 milliseconds. RB is faster for the OpenCL case in more than 83% of the cases, but the difference between R and RB is approximately up to 5.26 milliseconds. Nevertheless, in the CUDA case, R is the best option in 86% of the cases, with up to 0.63 milliseconds faster than RB.

In general, the best option for the block size is 16×8 for the smaller datasets (less than 40 MB), 16×8 or 8×8 for intermediate volumes (around 150 MB), and 4×4 or 8×4 for the larger volumes (around 600 MB), in almost all cases. As it can be observed, the best results tend to be block configurations with a width/height ratio of 1 : 1 or 1 : 2. For smaller volumes, selecting a high number of threads per block gives better performance, while when the volume gets larger, a better performance is obtained with fewer threads per block.

Now, taking the results obtained from the fragment shader, and the best results between R and RB for compute shader, OpenCL, and CUDA, we can obtain some interesting comparisons. For almost all volumes, compute shader shows the best results. It is from $1.03\times$ to $1.85\times$ faster than the fragment shader, from $1.5\times$ to $7.1\times$ faster than OpenCL, and from $1.05\times$ to $1.3\times$ faster than CUDA. The second best performance is obtained by CUDA, which is from $1.1\times$ to $5.3\times$ faster than OpenCL. Nevertheless, the best option between fragment shader and CUDA varies depending on the size of the volume and the transfer function. CUDA is a better option with larger volumes and with opaque transfer functions (TF1), whereas fragment shader presents better results for smaller volumes and semi-transparent transfer functions (TF2). Furthermore, the worst performance is shown by OpenCL, which is only better with respect to the fragment shader in some of the volumes at their maximum resolution.

Finally, an additional test was conducted to measure the performance of each parallel API with a diffuse illuminated volume ray casting. The results can be depicted in Table 4 and Figure 8. We only chose the first transfer function (TF1) for all volumes respectively, as these transfer functions render an opaque surface.

Table 4: Performance comparison of fragment shader, compute shader, OpenCL, and CUDA, using lighting

			Fragment	Compute R	Compute RB	OpenCL R	OpenCL RB	CUDA R	CUDA RB
male 512 × 512 × 1245	TF1	Time	195.01	66.34	65.86	139.81	139.41	80.25	81.15
		Size		4 × 2	4 × 2	4 × 4	4 × 4	4 × 4	4 × 4
male 512 × 256 × 622	TF1	Time	34.27	19.78	19.61	34.48	34.09	19.82	20.25
		Size		8 × 4	8 × 4	8 × 8	8 × 8	8 × 4	8 × 4
male 256 × 256 × 311	TF1	Time	5.88	5.25	5.09	17.06	16.65	5.92	6.16
		Size		8 × 8	8 × 8	16 × 8	16 × 8	8 × 8	8 × 8
male 128 × 128 × 311	TF1	Time	3.56	3.78	3.59	13.31	12.84	3.58	3.67
		Size		16 × 8	8 × 8	16 × 8	16 × 8	16 × 8	16 × 8
male 128 × 64 × 155	TF1	Time	2.02	1.93	1.78	8.47	8.30	2.00	2.12
		Size		16 × 8	16 × 8	16 × 8	16 × 8	16 × 8	16 × 8
lemni 512 × 512 × 1058	TF1	Time	137.74	54.62	55.05	110.73	110.19	62.44	63.43
		Size		4 × 4	4 × 4	8 × 4	8 × 4	4 × 4	4 × 4
lemni 512 × 256 × 529	TF1	Time	20.06	15.21	14.83	30.73	30.54	17.19	17.48
		Size		8 × 4	8 × 4	8 × 8	8 × 8	8 × 4	8 × 4
lemni 256 × 256 × 264	TF1	Time	5.25	5.00	4.80	17.64	17.30	5.71	5.80
		Size		8 × 8	8 × 8	16 × 8	16 × 8	8 × 8	16 × 8
lemni 128 × 128 × 264	TF1	Time	3.46	3.35	3.25	13.41	13.08	3.45	3.54
		Size		16 × 8	16 × 8	16 × 8	16 × 8	16 × 8	16 × 8
lemni 128 × 64 × 132	TF1	Time	2.10	1.92	1.82	9.19	8.92	2.10	2.23
		Size		16 × 8	16 × 8	16 × 8	16 × 8	16 × 8	16 × 8
beetle 832 × 832 × 494	TF1	Time	250.27	122.03	121.16	209.04	208.45	130.43	131.68
		Size		4 × 4	4 × 4	8 × 8	8 × 8	8 × 4	8 × 4
beetle 416 × 416 × 494	TF1	Time	75.90	51.97	51.71	98.64	98.45	57.99	58.18
		Size		8 × 8	8 × 4	8 × 8	8 × 8	8 × 8	8 × 8
beetle 416 × 208 × 247	TF1	Time	16.17	18.05	17.60	60.79	60.80	17.85	17.81
		Size		8 × 8	8 × 8	16 × 8	16 × 8	16 × 8	16 × 8
beetle 208 × 208 × 123	TF1	Time	8.88	8.39	8.29	36.92	37.08	8.93	9.10
		Size		16 × 8	16 × 8	16 × 8	16 × 8	16 × 8	16 × 8
beetle 104 × 104 × 123	TF1	Time	5.37	5.10	4.94	22.76	22.73	5.34	5.45
		Size		16 × 8	16 × 8	16 × 8	16 × 8	16 × 8	16 × 8

The results of this test show a similar behavior as the previous test. As before, some values are highlighted to indicate the best result between R and RB, for each implementation, dataset, and transfer function. In the compute shader case, RB is faster than R in 93.3% of the cases with little differences in the execution time. RB is faster for the OpenCL case in 86.6% of the cases on the table, again with little differences. Nevertheless, in the CUDA case, R is the best option in 93.3% of the cases, with little differences.

For smaller volume sizes (less than 40 MB), the best block configuration is 16×8 for almost all cases. In

Figure 8: Comparison of the execution time of fragment shader, compute shader, OpenCL, and CUDA using lighting

the case of intermediate volume sizes (around 150 MB), the best configurations are 8×4 , 8×8 , and 16×8 . Finally, for the larger volume sizes (around 600 MB) the best options are 4×2 , 4×4 , and 8×4 . The only exception with larger volumes is the OpenCL case, where the best option is 8×8 . As in the previous test, block configurations with an $1:1$ or $1:2$ ratio tend to be the best choice, with a higher number of threads per block for smaller volumes, and a fewer number of threads per block for larger volumes.

Comparing the best results between RB and R for each parallel API, and the fragment shader, we found that compute shader has the fastest results among all the implementations, with the exceptions of the male $128 \times 128 \times 311$ with respect to fragment shader and CUDA, and the beetle $416 \times 208 \times 247$ with respect to fragment shader. As before, CUDA is the second best option, being between $1.5\times$ and $4.2\times$ faster than OpenCL, and faster than fragment shader in some cases. Fragment shader shows better performance than OpenCL in almost all cases, except with the volumes at maximum resolution, and has better performance than CUDA in some smaller volume cases.

Finally, the increment in execution time for the addition of the diffuse lighting depends of the API. The increment for fragment shader is between $1.3\times$ and $2.8\times$, for compute shader is between $1.3\times$ and $1.7\times$, for OpenCL is between $1.08\times$ to $2.3\times$, and for CUDA is between $1.2\times$ to $1.7\times$. As can be observe, the overhead for lighting calculation is lower for compute shader and CUDA, than for fragment shader and OpenCL. Furthermore, we can observe than the increments for the male dataset are higher than the ones for the lemni dataset, for almost all cases. We do not obtain conclusive results comparing the beetle dataset with the other two, as the results varies depending of the API and size of the volumes.

6 Conclusions and Future Work

In this paper, we presented a performance comparison of volume rendering implementations using fragment shader, compute shader, OpenCL, and CUDA. The implementations were tested with three datasets, with different volume resolutions, from smaller volumes (less than 40 MB), through intermediate volumes (around 150 MB), to larger volumes (around 600 MB). Each volume was tested with different transfer functions and block sizes for the GPGPU implementations (4 to 128 threads per block). As expected, the performance of our ray casting implementations decreases with larger volumes, and with semi-transparent transfer function.

For the tests we conclude that compute shader implementation represents the best option to implement a basic volume ray caster in general. Compute shader shows the best results for almost all tested cases, being between $1.03\times$ and $2.9\times$ faster than fragment shader, between $1.5\times$ and $7.1\times$ faster than OpenCL, and between $1.01\times$ and $1.3\times$ faster than CUDA. The following best option is between fragment shader and CUDA. CUDA shows better results than fragment shader for almost all cases of larger and intermediate volumes, with a speed improvement from $1.1\times$ to $2.4\times$. Whereas, in smaller volumes there is not a global winner between fragment shader and CUDA. Finally, OpenCL has the worst performance for almost all cases which is between $1.01\times$ and $7.1\times$ times slower than any other API. The only exception is for volumes at its

highest resolutions, where OpenCL is up to $1.4\times$ faster than fragment shader.

Furthermore, an additional test was performed to measure the overhead of calculating diffuse illumination in each tested volume. The addition of the illumination makes the rendering execution time to be up to $2.8\times$ times slower than the normal rendering, as it has more calculation and more texture fetches. The increment in the execution time for fragment shader is up to $2.8\times$, for compute shader is up to $1.7\times$, for OpenCL is up to $2.3\times$, and for CUDA is up to $1.7\times$. Notice that the impact of the lighting calculation is smaller for compute shader and CUDA, being consistent with the results without lighting, where they were also the fastest approaches.

We also found that there is not a global winner in terms of execution time between ray/volume intersection methods (rasterization and ray/box). Nevertheless, the ray/box intersection presents better performance for most of the cases with compute shader and OpenCL, while the raster intersection test presents a better performance for most of the cases with CUDA. Thus, the best overall results of our test were obtained by combining compute shader with ray/box intersection test.

In our tests, we observed that the best block size varies with the volume size. For larger volume datasets, a configuration of, for example 4×4 , 8×4 , and 4×2 , are the best choices. It is worth notice the block configurations of 4×4 and 4×2 , as the number of threads is smaller than the CUDA warp size (32 threads). Theoretically, with these configurations, half or more than the half of the threads of the warps will be idle, so there will be a high amount of unused processing power, which should be inefficient. While the volume size becomes bigger, a configuration with more threads per blocks gives better results. For example, 8×8 or 16×8 for volumes up to around 150 MB. Block configurations with an 1 : 1 or 1 : 2 ratio present better results than the ones with rectangular aspect ratios, which prove that configurations with an 1 : 1 or 1 : 2 ratio makes more coalesce accesses to the volume texture, and a thread execution with less divergence. With block configurations with rectangular aspect ratios, as the threads fetches data from distant areas of the volume that could represent different materials, it is possible that a high number of threads do an early ray termination, which ends up in a high number of idle threads waiting for other threads in the same block to finish execution. Furthermore, for larger volumes, the best option is a fewer number of threads per block (e.g. 16 or 32), while for smaller volumes, the best option is a higher number of threads per block (e.g. 64 or 128). It may be related to cache performance, since larger block sizes require to access larger volume areas at the same time. For smaller volumes, those required voxels could be in the GPU cache, whereas in larger volumes it is more challenging for caching manager. The same cache performance problem could be seen in block configurations with less threads than the warp size for larger volumes. With larger volumes, each thread would fetch distant areas of the volumes, generating cache misses, that could end up in a serial resolution of the fetches. For that reason, even a number of threads fewer than the warp size could be beneficial, as a massive cache miss would be avoided for a warp.

As future work, it would be interesting to test the implementations in a newer GPU, with more capabilities, especially for newer versions of OpenCL and CUDA. Furthermore, as is usual in volume rendering, many rays may be early-terminated due to opacity accumulation; thus, using a method like persistent threads [36] could be beneficial for opaque transfer functions, and the speedup obtained should be studied.

References

- [1] Preim, Bernhard and Botha, Charl P., *Visual Computing for Medicine: Theory, Algorithms, and Applications*, 2nd ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2013.
- [2] J. Kruger and R. Westermann, "Acceleration Techniques for GPU-based Volume Rendering," in *Proceedings of the 14th IEEE Visualization 2003 (VIS '03)*, ser. VIS '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 38-. [Online]. Available: <https://doi.org/10.1109/visual.2003.1250384>
- [3] T. L. Kay and J. T. Kajiya, "Ray Tracing Complex Scenes," *SIGGRAPH Computer Graphics*, vol. 20, no. 4, pp. 269–278, Aug. 1986. [Online]. Available: <https://doi.org/10.1145/15886.15916>
- [4] J. Beyer, M. Hadwiger, and H. Pfister, "State-of-the-Art in GPU-Based Large-Scale Volume Visualization," *Computer Graphics Forum*, May 2015. [Online]. Available: <https://doi.org/10.1111/cgf.12605>
- [5] K. Group. OpenCL. [Accessed: 28-06-2017]. [Online]. Available: <http://www.khronos.org/opencl>
- [6] Nvidia. CUDA. [Accessed: 28-06-2017]. [Online]. Available: http://la.nvidia.com/object/cuda_home_new_la.html
- [7] K. Group. OpenGL. [Accessed: 28-06-2017]. [Online]. Available: <http://www.khronos.org/opengl>

- [8] D. Wolff, *OpenGL 4 Shading Language Cookbook*, 2nd ed. Packt Publishing Ltd., December 2013.
- [9] B. Cabral, N. Cam, and J. Foran, "Accelerated Volume Rendering and Tomographic Reconstruction Using Texture Mapping Hardware," in *Proceedings of the 1994 Symposium on Volume Visualization*, ser. VVS '94. New York, NY, USA: ACM, 1994, pp. 91–98. [Online]. Available: <http://doi.acm.org/10.1145/197938.197972>
- [10] T. J. Cullip and U. Neumann, "Accelerating Volume Reconstruction With 3D Texture Hardware," University of North Carolina at Chapel Hill, Chapel Hill, NC, USA, Tech. Rep., 1994.
- [11] S. Roettger, S. Guthe, D. Weiskopf, T. Ertl, and W. Strasser, "Smart Hardware-accelerated Volume Rendering," in *Proceedings of the Symposium on Data Visualisation 2003*, ser. VISSYM '03. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2003, pp. 231–238.
- [12] M. Hadwiger, C. Sigg, H. Scharsach, K. Bühler, and M. Gross, "Real-Time Ray-Casting and Advanced Shading of Discrete Isosurfaces," *Computer graphics forum*, vol. 24, no. 3, pp. 303–312, Sep. 2005. [Online]. Available: <https://doi.org/10.1111/j.1467-8659.2005.00855.x>
- [13] P. Lacroute, "Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transformation," Standfor University, Tech. Rep., 1995.
- [14] M. Garland, S. L. Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, and V. Volkov, "Parallel Computing Experiences with CUDA," *Micro, IEEE*, vol. 28, no. 04, pp. 13–27, September 2008. [Online]. Available: <http://dx.doi.org/10.1109/MM.2008.57>
- [15] J. Shen, J. Fang, H. Sips, and A. L. Varbanescu, "An Application - Centric Evaluation of OpenCL on Multi-Core CPUs," *Parallel Computing*, vol. 39, no. 12, pp. 834–850, December 2013. [Online]. Available: <https://doi.org/10.1016/j.parco.2013.08.009>
- [16] N. Bombieri, S. Vinco, V. Bertacco, and D. Chatterjee, "SystemC Simulation on GP-GPUs: CUDA vs. OpenCL," in *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware / Software Codesign and System Synthesis (CODES+ISSS '12)*. Tampere, Finland: ACM, October 2012, pp. 343–352. [Online]. Available: <https://doi.org/10.1145/2380445.2380500>
- [17] K. Karimi, N. Dickson, and F. Hamze, "A Performance Comparison of CUDA and OpenCL," *ArXiv e-prints*, p. 12, May 2011.
- [18] K. Komatsu, K. Sato, Y. Arai, K. Koyama, H. Takizawa, and H. Kobayashi, "Evaluating Performance and Portability of OpenCL Programs," in *Proceedings of the Fifth international Workshop on Automatic Performance Tuning (iWAPT '10)*, Berkeley, USA, June 2010.
- [19] C.-L. Su, P.-Y. Chen, C.-C. Lan, L.-S. Huang, and K.-H. Wu, "Overview and Comparison of OpenCL and CUDA Technology for GPGPU," in *2012 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS)*, Kaohsiung, Taiwan, December 2012, pp. 448–451. [Online]. Available: <https://doi.org/10.1109/APCCAS.2012.6419068>
- [20] N. Satish, M. Harris, and M. Garland, "Designing Efficient Sorting Algorithms for Manycore GPUs," in *IEEE International Symposium on Parallel & Distributed Processing, 2009 (IPDPS '09)*, Rome, Italy, May 2009, pp. 1–10. [Online]. Available: <https://doi.org/10.1109/ipdps.2009.5161005>
- [21] R. Sachetto Oliveira, B. M. Rocha, R. M. Amorim, F. O. Campos, J. Meira, Wagner, E. M. Toledo, and R. W. dos Santos, "Comparing CUDA, OpenCL and OpenGL Implementations of the Cardiac Monodomain Equations," in *Parallel Processing and Applied Mathematics*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, September 2012, vol. 7204, pp. 111–120. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-31500-8_12
- [22] T. I. Vassilev, "Comparison of Several Parallel API for Cloth Modelling on Modern GPUs," in *Proceedings of the 11th International Conference on Computer Systems and Technologies and Workshop for PhD Students in Computing on International Conference on Computer Systems and Technologies*, ser. CompSysTech '10. Sofia, Bulgaria: ACM, June 2010, pp. 131–136. [Online]. Available: <http://doi.acm.org/10.1145/1839379.1839403>
- [23] L. Marsalek, A. Hauber, and P. Slusallek, "High-Speed Volume Ray Casting with CUDA," in *Interactive Ray Tracing, 2008. RT 2008. IEEE Symposium on*, Aug 2008, pp. 185–185. [Online]. Available: <https://doi.org/10.1109/rt.2008.4634648>

- [24] N. Schubert and I. Scholl, "Comparing GPU-based Multi-Volume Ray Casting Techniques," *Computer Science - Research and Development*, vol. 26, no. 1-2, pp. 39–50, February 2011. [Online]. Available: <http://dx.doi.org/10.1007/s00450-010-0141-1>
- [25] J. Fangerau and S. Krömker, *Facing the Multicore-Challenge: Aspects of New Paradigms and Technologies in Parallel Computing*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, ch. Parallel Volume Rendering Implementation on Graphics Cards Using CUDA, pp. 143–153.
- [26] P. Kumar and A. Agrawal, *Intelligent Interactive Technologies and Multimedia: Second International Conference, IITM 2013, Allahabad, India, March 9-11, 2013. Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, ch. CUDA Based Interactive Volume Rendering of 3D Medical Data, pp. 123–132.
- [27] G. Kiss, E. Steen, J. Asen, and H. Torp, "GPU Volume Rendering in 3D Echocardiography: Real-time Pre-Processing and Ray-Casting," in *Ultrasonics Symposium (IUS), 2010 IEEE*, October 2010, pp. 193–196. [Online]. Available: <http://dx.doi.org/10.1109/ULTSYM.2010.5935485>
- [28] A. Weinlich, B. Keck, H. Scherl, M. Kowarschik, and J. Hornegger, "Comparison of High-Speed Ray Casting on GPU using CUDA and OpenGL," in *Proceedings of the First International Workshop on New Frontiers in High-performance and Hardware-aware Computing*, vol. 1, November 2008, pp. 25–30.
- [29] Z. Shi, C. J. Yi, and X. C. Hua, "Algorithm of Ray Casting Volume Rendering Based on CUDA," in *The 2nd International Conference on Industrial Application Engineering 2014 (ICIAE '14)*, Changshu, China, March 2014. [Online]. Available: <http://dx.doi.org/10.12792/iciae2014.011>
- [30] J. Mensmann, T. Ropinski, and K. Hinrichs, "An Advanced Volume Raycasting Technique using GPU Stream Processing," in *5th International Conference on Computer Graphics Theory and Applications (GRAPP '10)*, Anger, France, May 2010. [Online]. Available: <http://dx.doi.org/10.5220/0002843201900198>
- [31] M. Levoy, "Display of Surfaces from Volume Data," *IEEE Computer Graphics and Applications*, vol. 8, no. 3, pp. 29–37, May 1988. [Online]. Available: <https://doi.org/10.1109/38.511>
- [32] P. Sabella, "A Rendering Algorithm for Visualizing 3D Scalar Fields," *SIGGRAPH Computer Graphics*, vol. 22, no. 4, pp. 51–58, Jun. 1988. [Online]. Available: <https://doi.org/10.1145/378456.378476>
- [33] T. Aila, S. Laine, and T. Karras, "Understanding the Efficiency of Ray Traversal on GPUs - Kepler and Fermi Addendum," NVIDIA Research, Tech. Rep., 2012.
- [34] U. N. L. of Medicine. Visible Human Project. [Accessed: 28-06-2017]. [Online]. Available: http://www.nlm.nih.gov/research/visible/visible_human.html.
- [35] Meister Eduard Gröller and Georg Glaeser and Johannes Kastner. (2005) Stag beetle. [Accessed: 28-06-2017]. [Online]. Available: <https://www.cg.tuwien.ac.at/research/publications/2005/dataset-stagbeetle/>
- [36] T. Aila and S. Laine, "Understanding the Efficiency of Ray Traversal on GPUs," in *Proceedings of the Conference on High Performance Graphics 2009*, ser. HPG '09. New York, NY, USA: ACM, 2009, pp. 145–149. [Online]. Available: <https://doi.org/10.1145/1572769.1572792>

Appendix

The Table 5 depicts an example of the conducted test for only one dataset and one parallel API. Here every block configuration is tested, and the average execution time to generate a frame in milliseconds is shown. A similar table was generated for every dataset and parallel API tested to determine the best block configuration to render that volume. In this case, we can observe that the opaque transfer functions (TF1) is always faster than the transparent transfer function (TF2), and adding illumination always makes the rendering process slower. Furthermore, block configurations with an 1 : 1 or 1 : 2 ratio, always represent better options than configurations with rectangular aspect ratios. Best block configuration is 4×4 for the non-illuminated volumes, and 4×2 for the illuminated volumes.

Table 5: Time per frame for rendering the dataset male $512 \times 512 \times 1245$ using compute shader

Block	TF	R	RB	Lit R	Lit RB
4×1	TF1	108.53	107.50	118.89	116.81
	TF2	184.33	184.66		
2×2	TF1	102.13	102.45	111.93	109.83
	TF2	174.43	174.44		
8×1	TF1	64.08	63.82	82.19	81.33
	TF2	112.37	111.66		
4×2	TF1	58.46	57.91	66.34	65.87
	TF2	100.69	100.61		
16×1	TF1	66.92	67.11	166.13	170.38
	TF2	124.41	124.57		
8×2	TF1	41.69	41.36	96.59	97.95
	TF2	83.97	83.12		
4×4	TF1	37.91	36.99	80.28	81.12
	TF2	68.60	68.21		
32×1	TF1	107.43	107.80	282.62	279.19
	TF2	195.19	195.29		
16×2	TF1	77.30	77.53	191.44	190.53
	TF2	144.66	144.36		
8×4	TF1	59.37	59.74	146.71	147.27
	TF2	119.44	119.00		
64×1	TF1	120.91	121.23	330.48	322.41
	TF2	212.11	212.49		
32×2	TF1	119.05	119.61	326.05	319.22
	TF2	209.16	209.67		
16×4	TF1	93.69	94.10	241.13	234.87
	TF2	167.24	167.22		
8×8	TF1	75.95	75.65	191.10	188.76
	TF2	132.71	132.59		
128×1	TF1	123.33	123.69	348.21	347.23
	TF2	214.58	215.03		
64×2	TF1	121.99	123.14	363.13	362.94
	TF2	212.91	213.61		
32×4	TF1	121.30	122.22	363.70	363.79
	TF2	211.75	212.05		
16×8	TF1	97.40	99.27	273.41	272.89
	TF2	175.83	178.59		