Volume Ray Casting using Different GPU based Parallel APIs

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

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.**

***Index Terms*—GPU, OpenGL, OpenCL, CUDA, Performance Comparison, Volume Ray Casting, Parallel APIs, Compute Shader**

I. 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 programmables, and the programmable stages are mainly limited to process geometries, vertexes and fragments [7] . Nevertheless, nowadays OpenGL® 4.5 allows the general purpose computing with the compute shader [8]. One of the advantage 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 volume ray casting with four implementation, 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 ways 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 follow: Section II makes a survey of the related work. Section III briefly describes volume ray casting, showing how this method is optimized with parallel APIs. Then, Section IV shows some details of the volume ray casting implementations used for testing. In Section V, the test datasets and results are presented. Finally, Section VI discusses the conclusions and future work.

II. 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 the use of 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 [11] and Kruger and Westermann [2] were among the first to introduce this method, using a multi-pass approach. Later, authors like Hadwiger [12] implemented a single pass ray casting.

978-1-5090-1633-4/16/$31.00 ©2016 IEEE

To know which volume rendering method is better, the performance of different methods has been studied; for example, by comparing shear-warp with ray casting and splatting [1]. 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. For example, Garland et al. [13] 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. [14] 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. For example, OpenCL and CUDA were compared in different studies with SystemC simulation [15], Monte Carlo simulation of a quantum spin system [16], and several other programs [17][18]. 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 [19] 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 [20] and cloth modelling [21]. Those studies showed that there is not a unique best API choice for all problems, since the best API choice varies problem by problem.

Some studies have focused in measuring the performance of volume ray casting using a parallel API. Marsalek et al. [22] demonstrated proof of concept of a simple CUDA ray caster. Schubert and Scholl [23] 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 [24] and [25]. Fangerau and Kromer [24] 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. [26] 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. For example, Weinlich, et al. [27] 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. [28] 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. [29] 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 compared 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.

III. 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 [30]. This technique simulates the interaction between the light and a participating media (the volume), and is modeled with the Equation 1.

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

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 [31]. 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 Fig. 1.

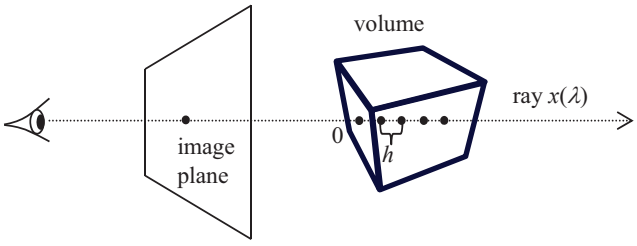


Fig. 1. Calculate the color of a pixel of the image from the contributions of color and opacity of the equidistant samples of the ray traversal of the volume.

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.

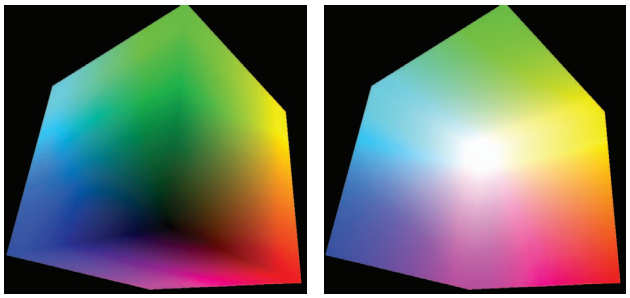
$$C \approx \sum_{i=0}^{n-1} \prod_{j=0}^{i-1} (1 - \alpha_j)\alpha_i c_i \tag{2}$$

As volume ray casting evaluates the volume rendering equation directly, it achieves good visual results. 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 and important improvement 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].

A. 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 Fig. 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 the ray (entry point), and with front face culling to obtain the last hit (exit point). With this two values it is possible to calculate the origin of the ray (first hit, Fig. 2a), end of the ray (last hit, Fig. 2b), and direction (last hit minus first hit, Fig. 2c).



(a) (b)

(c)

Fig. 2. Images obtained by rasterization that contains: (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).

B. 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 [32]. 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 bbox 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.

IV. 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, using 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.

A. 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 (`lasHit`) is fetched with the function `texelFetch`. With `first` and `lasHit`, 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 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`.

B. Compute shader

Listing 3 shows the implementation of the volume ray casting using the 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 with 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 the 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.

C. 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 binded 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 its used 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 variable `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.

D. CUDA

Finally, the CUDA 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 = muchar4(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 the 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 the 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.

V. 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.

A. 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. That video card supports OpenGL® 4.5, OpenCL 1.2, and CUDA with compute capability 3.0.

All test datasets were rendered at a grazing angle with the same rotation. We selected four datasets for evaluation (see Table I). For the first 3 datasets: radiography of a foot (Fig. 3), angiography with an aneurysm (Fig. 5) and visible male CT from the Visible Human project [33] (Fig. 4) we used 4 different transfer functions. The objective of the first transfer function (a) is to show a semi-transparent low-density tissue, and opaque bones (foot, male, angiography TF1). Thus, the rays traverse through the semi-transparent area until a full opaque voxel is reached. For the second transfer function (b), we just define the skin as opaque (foot, male, angiography TF2). Thus, most of the rays will stop after hitting the body skin, and ray casting should finish faster. For the third transfer function (c), only the bones are visible, and thus, those rays that hit the bones will be early-terminated (foot, male, angiography TF3). The last transfer function (d), is semi-transparent (foot, male, angiography TF4). Thus, early ray termination will be unlikely. The last dataset (Fig. 6) was generated with a implicit equation. In this case, we only use 2 transfer functions: (a) full opacity at first hit of some particular surface (lemniscate TF1) and (b) many semi-transparent surface (lemniscate TF2).

TABLE I
Size of the Datasets.

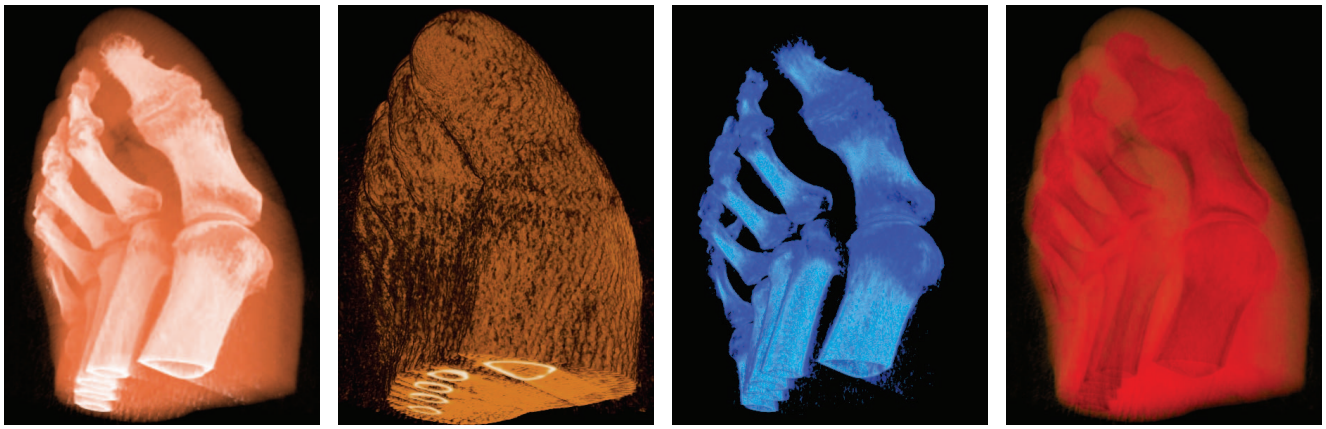
	foot	angiography	male	lemniscate
Width (voxels)	256	512	512	512
Height (voxels)	256	512	512	512
Depth (voxels)	256	1559	1245	1058
Bits per voxel	8	16	16	16
Volume size (MB)	16	779	622	529

B. 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 number of threads: 4, 8, 16, 32, 64, 128; and for each size, different configurations were tested (see Table II). All of these configurations are considered to test with squared and rectangular combinations. Other block sizes and configurations where 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 above. 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.

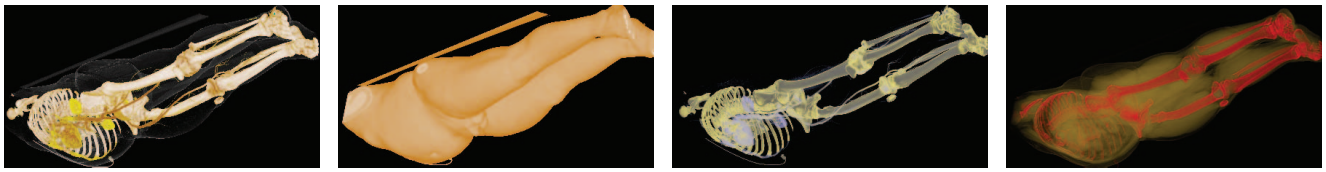
Furthermore, tests were performed using rasterization, and ray/box intersection test, for all datasets, all transfer functions, and with a fixed screen resolution of 1024×768. Each pixel of the screen will be mapped into a thread, and the block size will indicate how those threads are grouped. If for 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.

Then, the results obtained in the first test are compared with the use of the fragment shader. Using the fragment shader there is not need of choosing a block size. Since the results depends of the volume size, we include an additional test with different downsampled versions of the angiography dataset, to check if the results obtained in the previous tests are consistent.



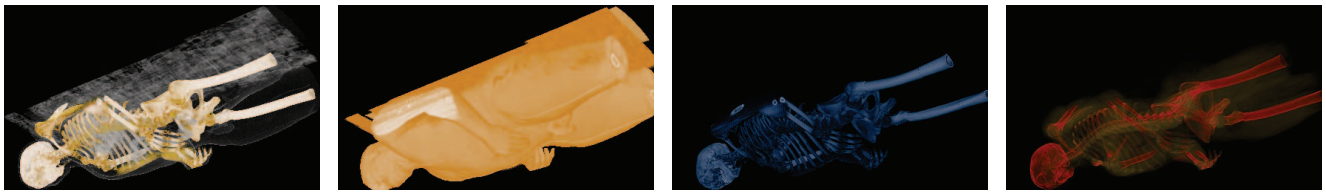
(a) Foot TF1. (b) Foot TF2. (c) Foot TF3. (d) Foot TF4.

Fig. 3. Render of the foot dataset with 4 different transfer functions.



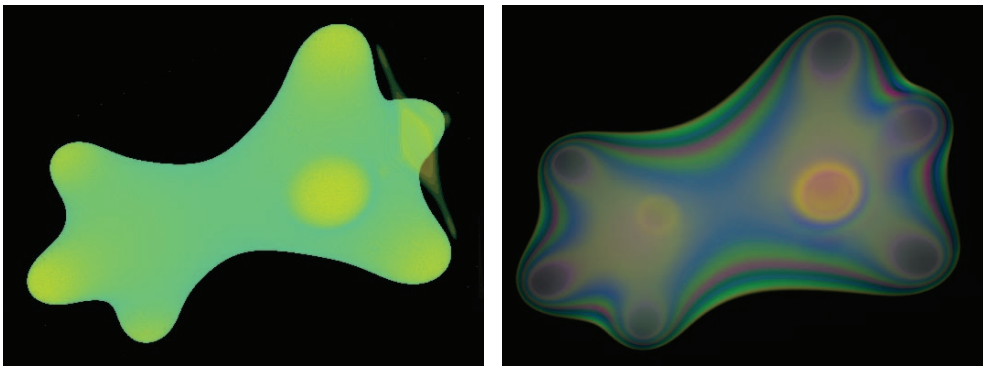
(a) Male TF1. (b) Male TF2. (c) Male TF3. (d) Male TF4.

Fig. 4. Render of the angiography dataset with 4 different transfer functions.



(a) Angiography TF1. (b) Angiography TF2. (c) Angiography TF3. (d) Angiography TF4.

Fig. 5. Render of the male dataset with 4 different transfer functions.



(a) Lemniscate TF1. (b) Lemniscate TF2.

Fig. 6. Render of the lemniscate dataset with 2 different transfer functions.

TABLE II
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

C. Results

First, the results of the execution time in milliseconds per frame is presented in Table III for the compute shader, OpenCL, and CUDA. Those results are shown in milliseconds per frame using two methods: rasterization (Raster), and ray/box intersection test (R/B). Due to lack of space, only the time for the best block size configuration are presented alongside with the corresponding block size configuration. For measuring the Raster and the R/B test, 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 (Raster and R/B), including and excluding the display of the final image into the screen, to measure the display overhead. Display overhead is bounded by up to 0.5 ms, which represents less than 0.5% of the execution time in all cases, indicating that this overhead is negligible.

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 Raster and R/B, for each implementation, dataset, and transfer function. In the compute shader case, the R/B is always faster than Raster in up to 1.04 millisecond. R/B is also faster for the OpenCL case in more than 70% of the cases on the table, but the difference between Raster and R/B is approximately less than 1.5 milliseconds. This difference is even smaller in CUDA, with approximately less than 0.5 milliseconds, but this time the rasterization intersection test is faster in 70% of the cases.

In general, the best option for the block size is 16×8 for the smaller dataset (foot), and 4×4 for the other volumes, in almost all cases. The only exception is the OpenCL R/B, where the best configuration is 8×8 for almost all cases.

As it can be observed the best results tend to be squared block configurations. For smaller volumes, selecting a higher number of threads per block gives better performance, while for larger volumes, a fewer number of threads per block gives better performance.

Table IV and Figure 7 present the performance comparison between fragment shader and the best results obtained in the previous test for compute shader, OpenCL, and CUDA implementations (highlighted results in Table III). As it can be observed, compute shader always presents the best performance. In the angiography, male, and lemniscate datasets, compute shader has a speed improvement from 46% to 92% with respect of fragment shader, from 31% to 67% with respect of OpenCL, and from 7% to 30% with respect of CUDA. This tendency changes with the foot dataset, as it is the smallest dataset with 16 MB. Considering the foot dataset, compute shader presents the best performance, followed by fragment shader, CUDA, and finally OpenCL. OpenCL shows the worst performance of all the implementations (around 391% to 497% slower).

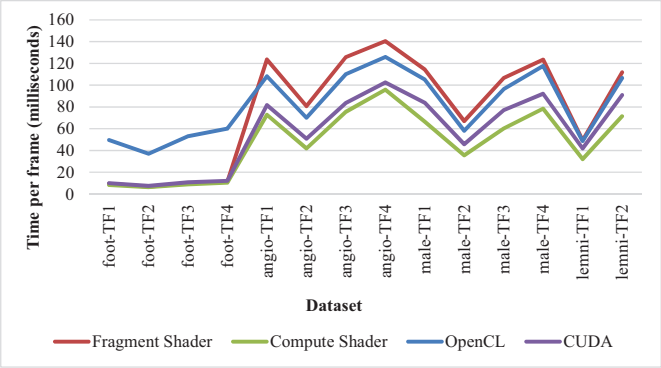


Fig. 7. Comparison of the execution time of all the tested APIs.

Finally, an additional test was conducted to measure the performance of each technology with the same volume but with different downsampled sizes. We chose the angiography dataset because it is the larger dataset. We selected the transfer function TF4 due to its transparency. All the block sizes shown in Table II were tested, with Raster and R/B intersection test. The results can be depicted in Table V and Figure 8.

This test confirmed the behavior obtained before. Compute shader shows the fastest results among all the implementations, followed by CUDA and then OpenCL. Fragment shader performance decreases faster than the other implementations with larger volumes ($512 \times 512 \times 779$ and $512 \times 512 \times 1559$), becoming the worst option for these cases. However, for the smaller volume sizes ($64 \times 64 \times 194$, $128 \times 128 \times 389$, and $256 \times 256 \times 779$) fragment shader represents the second best option.

It can be noted that for smaller volume sizes, the best block configuration is 16×8 for almost all cases. However, for larger sizes, compute shader and CUDA show better results with a block configuration of 4×4, and OpenCL shows better results with 8×4 and 8×8. Finally, for intermediate sizes, the best configurations are 8×8 and 8×4. As in the previous tests, squared block configurations present better results than the rectangular ones. Also, testing with smaller volumes, we obtain a better performance with a higher number of threads per block, while testing with larger volumes, we get a better performance with a lower number of threads per blocks.

TABLE III
PERFORMANCE COMPARISON OF VOLUME RAY CASTING IMPLEMENTED IN COMPUTE SHADER, OPENCL AND CUDA.

		foot				angio				male				lemni	
		TF1	**TF2**	**TF3**	**TF4**	**TF1**	**TF2**	**TF3**	**TF4**	**TF1**	**TF2**	**TF3**	**TF4**	**TF1**	**TF2**
Compute Raster	**Size**	16×8	16×8	16×8	16×8	4×4	4×4	4×4	4×4	4×4	4×4	4×4	4×4	4×4	4×4
	Time	8.45	6.49	9.04	10.42	73.82	42.25	76.27	96.92	66.85	35.84	**60.17**	79.29	32.37	71.80
Compute R/B	**Size**	16×8	16×8	16×8	16×8	4×4	4×4	4×4	4×4	4×4	4×4	4×4	4×4	4×4	4×4
	Time	**8.34**	**6.33**	**8.89**	**10.41**	**72.85**	**41.87**	**75.50**	**95.87**	**66.50**	**35.62**	**60.17**	**78.34**	**32.03**	**71.40**
OpenCL Raster	**Size**	16×8	16×8	16×8	16×8	4×4	4×4	4×4	4×4	4×4	4×4	4×4	4×4	4×4	4×4
	Time	49.67	37.12	53.20	**60.10**	109.61	71.26	111.50	127.12	**105.01**	58.13	**96.45**	**117.65**	48.78	106.64
OpenCL R/B	**Size**	8×8	8×8	8×8	16×8	8×4	8×4	8×4	8×4	8×8	8×8	8×8	8×8	8×8	8×8
	Time	**49.60**	**37.03**	**53.10**	60.15	**108.20**	**70.05**	**110.06**	**125.92**	105.27	**58.06**	96.49	118.23	**48.59**	**106.61**
CUDA Raster	**Size**	16×8	16×8	16×8	16×8	4×4	4×4	4×4	4×4	4×4	4×4	4×4	4×4	4×4	4×4
	Time	**10.05**	**7.52**	**10.81**	**12.12**	81.74	**50.83**	83.77	103.04	**83.78**	**45.86**	**77.01**	92.35	**41.86**	**91.09**
CUDA R/B	**Size**	16×8	16×8	16×8	16×8	4×4	4×4	4×4	4×4	4×4	4×4	4×4	4×4	8×4	4×4
	Time	10.08	7.54	10.84	12.13	**81.64**	51.05	**83.63**	**102.51**	83.80	46.23	77.32	**92.16**	41.93	91.28

TABLE IV
PERFORMANCE COMPARISON OF FRAGMENT SHADER, COMPUTE SHADER, OPENCL, AND CUDA.

		Fragment Shader	**Compute Shader**	**OpenCL**	**CUDA**
foot	**TF1**	9.46	**8.34**	49.60	10.05
	TF2	6.80	**6.33**	37.03	7.52
	TF3	9.91	**8.89**	53.10	10.81
	TF4	11.20	**10.41**	60.10	12.12
angio	**TF1**	123.59	**72.85**	108.20	81.64
	TF2	80.75	**41.87**	70.05	50.83
	TF3	125.76	**75.50**	110.06	83.63
	TF4	140.46	**95.87**	125.92	102.51
male	**TF1**	114.31	**66.50**	105.01	83.78
	TF2	67.11	**35.62**	58.06	45.86
	TF3	106.58	**60.17**	96.45	77.01
	TF4	123.52	**78.34**	117.65	92.16
lemni	**TF1**	49.40	**32.03**	48.59	41.86
	TF2	111.84	**71.40**	106.61	91.09

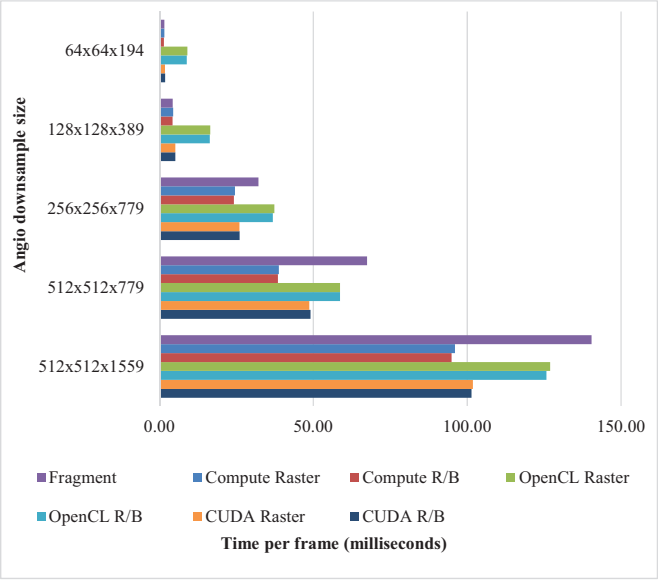


Fig. 8. Comparison of the execution time of fragment shader, compute shader, OpenCL, and CUDA for different sizes of the downsampled angiography dataset.

VI. 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 different datasets, from smaller volumes (16 MB) to larger volumes (around 600 MB), each volume 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 a larger size of the volume, 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 all tested cases. For volume datasets around 600 MB, it has a speed improvement from 46% to 92% with respect of fragment shader, from 31% to 67% with respect of OpenCL, and from 7% to 30% with respect of CUDA. It follows that CUDA is faster than OpenCL, and OpenCL is faster than fragment shader for these volumes. Only for the foot dataset (the smallest one with only 16 MB and 8 bits per voxel), the results change. In this special case, compute shader is still the best option, but now followed by CUDA, then fragment shader, and finally OpenCL.

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

TABLE V
PERFORMANCE COMPARISON OF ALL TESTED APIS USING DOWNSAMPLED REPRESENTATIONS OF THE ANGIOGRAPHY DATASET.

		Size				
		$512 \times 512 \times 1559$	$512 \times 512 \times 779$	$256 \times 256 \times 779$	$128 \times 128 \times 389$	$64 \times 64 \times 194$
GLSL	**Time**	140.39	67.49	32.18	4.30	1.53
Compute Raster	**Size**	4×4	4×4	8×4	8×8	16×8
	Time	96.04	38.78	24.57	4.43	1.54
Compute R/B	**Size**	4×4	4×4	8×4	8×8	16×8
	Time	**94.94**	**38.50**	**24.19**	**4.27**	**1.45**
OpenCL Raster	**Size**	8×4	8×8	8×8	16×8	16×8
	Time	126.99	58.68	37.36	16.47	9.05
OpenCL R/B	**Size**	8×4	8×8	8×8	16×8	16×8
	Time	125.74	58.69	36.84	16.33	8.84
CUDA Raster	**Size**	4×4	4×4	8×4	16×8	16×8
	Time	101.85	48.72	25.99	5.12	1.79
CUDA R/B	**Size**	4×4	4×4	8×4	16×8	16×8
	Time	101.36	49.07	26.04	5.14	1.84

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 4×4 is the best choice. While the volume size becomes smaller, a configuration with more threads per blocks gives better results. For example, 16×8 for volumes of up to 16 MB. Squared block configurations present better results than the rectangular ones, which prove that square configurations makes more coalesce accesses to the volume texture. Furthermore, for larger volumes, the best option is a fewer number of threads per block (e.g. 16), while for smaller volumes, the best option is a higher number of threads per block (e.g. 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.

As future work, it would be interesting to test the application in a newer GPU, with more capabilities, especially for newer versions of OpenCL and CUDA. As is usual in volume rendering, many rays may be early-terminated due to opacity accumulation; thus, using a method like persistent threads [34] could be beneficial for opaque transfer functions.

REFERENCES

[1] P. Lacroute, "Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transformation," Standfor University, Tech. Rep., 1995.

[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–.

[3] T. L. Kay and J. T. Kajiya, "Ray Tracing Complex Scenes," *SIGGRAPH Computer Graphics*, vol. 20, no. 4, pp. 269–278, Aug. 1986.

[4] J. Beyer, M. Hadwiger, and H. Pfister, "State-of-the-Art in GPU-Based Large-Scale Volume Visualization," *Computer Graphics Forum*, May 2015.

[5] K. Group. OpenCL. [Accedido: 02-07-2015]. [Online]. Available: http://www.khronos.org/opencl

[6] Nvidia. CUDA. [Accedido: 02-07-2015]. [Online]. Available: http://la.nvidia.com/object/cuda_home_new_la.html

[7] K. Group. OpenGL. [Accedido: 02-07-2015]. [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.

[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.

[13] 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.

[14] 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.

[15] 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.

[16] K. Karimi, N. Dickson, and F. Hamze, "A Performance Comparison of CUDA and OpenCL," *ArXiv e-prints*, p. 12, May 2011.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] M. Levoy, "Display of Surfaces from Volume Data," *IEEE Computer Graphics and Applications*, vol. 8, no. 3, pp. 29–37, May 1988.

[31] P. Sabella, "A Rendering Algorithm for Visualizing 3D Scalar Fields," *SIGGRAPH Computer Graphics*, vol. 22, no. 4, pp. 51–58, Jun. 1988.

[32] T. Aila, S. Laine, and T. Karras, "Understanding the Efficiency of Ray Traversal on GPUs - Kepler and Fermi Addendum," NVIDIA Research, Tech. Rep., 2012.

[33] U. N. L. of Medicine. Visible Human Project. [Accedido: 02-07-2015]. [Online]. Available: http://www.nlm.nih.gov/research/visible/visible_human.html.

[34] 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.