GPU Acceleration of Particle-based Volume Rendering using CUDA

Ding Zhongming 1), Naohisa Sakamoto 2)1), Yasuo Ebara 4), Koji Koyamada 3)

1) Graduate School of Electrical Engineering, Kyoto University
2) Center for the Promotion of Excellence in Higher Education, Kyoto University
3) KGT inc.
4) Academic Center for Computing and Media Studies, Kyoto University

ABSTRACT: In this paper, we apply Particle-based Volume Rendering (PBVR) technique using a current programmable GPU architecture. Recently, the increasing programmability of GPU offers an efficient method of SIMD parallel algorithm to solve the speed problem. Due to the each point or pixel can be calculated independently, we use programmable graphics hardware to delegate all expensive rendering tasks to the GPU. Here we apply on the popular programming architecture CUDA based on GeForce 8800 graphics unit. This approach allows enormous volume particles to be rendered in SIMD way instead of time-costing sequence processing so that the rendering speed can be accelerated. In this processing, each particle can be handled separately by one of the multi-processors in GPU. In this implementation, we introduce a non-confliction way to map the conventional algorithm onto CUDA calculation architecture efficiently. We apply CUDA in the programming framework as a general purpose GPU calculation for PBVR instead of using conventional GPU pipeline. All the rendering flow can be divided into three stages: the beforehand data arrangement, particle projection and sub-pixel processing. In order to evaluate the performance, we compare the frame rate of GPU accelerated PBVR with traditional CPU based approach.

KEY WORDS: GPU based volume rendering, particle-based volume rendering, CUDA

1. INTRODUCTION

Particle-based approaches are widely used in 3D scan data display and large scale particles visualization. Since the scale and complexity of volume datasets is increasing, there is a strong demand for volume rendering techniques capable of handling huge and complex volume datasets. To render such volume datasets with conventional volume rendering techniques, the calculations required for sorting and alpha blending can become a bottleneck.

Particle-based volume rendering (PBVR) technique can be effectively utilized to render various types of volume datasets, either regular or irregular comparing against ray-casting method [1]. In order to gain a proper rendering quality, large amount of particles need to be generated and projected for pixel processing. In the other hand, high quality result image can also be acquired by enhancing the sub-pixel level. However either of two stages costs large CPU processing time.

Recently, the increasing programmability of GPU offers an efficient method for SIMD parallel algorithm designing. Due to the each point or pixel can be calculated independently, we use programmable graphics hardware to delegate all expensive rendering tasks to the GPU. Here we implement on the popular programming architecture CUDA based on GeForce 8800 graphics unit.

The time cost of large scale data computing is always come up with heavy data access time waste. Since the local data processing speed of GPU multiprocessor is relatively faster than memory data fetch, designing an appropriate memory data arrangement become a main focus of our work.

Usually the SIMD parallel computing often brings a risk of memory access confliction which could drastically cumber the processing or even causes a crash. In this paper we also discuss about a foreground prediction sorting method to avoid confliction which is likely to happen in buffer writing.

2. OVERVIEW OF PBVR

Recently, volume rendering technique which uses particles as rendering primitives has received increasing attention and is a topic of ongoing research. Hopf and Ertl proposed the use of a hierarchical data structure in order to accelerate the visualization of scattered particle data [1]. This data structure enables fast sorting of semi-transparent clusters and may therefore trade rendering speed for image quality. The image is obtained by traversing this structure for each frame. In addition, the use of quantized relative coordinates reduced the memory consumption. In order to realize a semi-transparent effect, the technique requires alpha blending which requires the particles to be sorted according to their projected z-coordinates.

The proposed particle-based volume rendering technique represents the volume data as a set of particles. The particle density is derived from a user-specified transfer function and describes the probability that a particle is present at a given position. Since these
particles can be considered to be fully opaque, no alpha blending is required during the rendering calculation, which is advantageous in the distributed processing. Csébfalvi proposed a similar technique [2] that can be categorized as X-ray volume rendering, in which the number of particles on the image plane is simply counted, however the luminosity of the particles is not considered.

To generate the particle with brightness according to z-buffer depth, Sakamoto et al implemented particle-based volume rendering in 2007 [3]. This approach represents a 3D scalar field as a set of particles. In general, volume rendering techniques utilize an illumination model in which the 3D scalar field is characterized as a varying density emitter with a single level of scattering. This model is related to a particle model in which the size of the particle is small for the radius of the viewing ray, which is the pixel size. A conventional volume rendering technique models the density of particles, not the particles themselves. That is, a given scalar field is described as a continuous semi-transparent gel, and the accumulating order is important. This results in a considerable computational overhead.

We divide the PBVR technique into three phases: data arrangement, particle projection, and sub-pixel processing. The first phase is to construct a density field and to generate particles according to the density function. The second phase is to project particles onto an image plane. The third phase is to divide a single pixel into multiple sub-pixels so that a particle is stored as precisely as possible, and to calculate a final brightness value using sub-pixel values. In this proposal we map the second and third phases into GPU memory and calculation architecture.

3. PROGRAM FRAMEWORK

We assume that the scalar volume is sampled on random distributing particles. This is typical discrete representation of volume data in practice. In many related work of GPU based rendering, the programming pipeline is implemented in a fixed sequence of stages [5]: vertex processing and fragment processing. Since this pipeline is exclusively designed for polygonal rendering, volumetric primitives, such as tetrahedral or particle cannot be directly used. Usually some intricate mapping technique needs to be designed for volumetric rendering in this kind of graphic pipeline [7]. Many previous works implement the geometry set-up of volume rendering using shading language, such as Cg. This approach allows the fragment process and vertex process compositing the volume data through a texture mapping mode. Hence, volumetric object are forced to decompose into primitives supported by the graphics pipeline [6].

Instead of using the conventional pipeline decomposing volume data, we use CUDA as a general purpose GPU program approach for PBVR in a direct calculation style. Because CUDA provides a user specifiable data computing and memory accessing features, particles can be calculated in most readable program style and executed in SIMD way. All the rendering flow in GPU side can be divided into three stages: data arrangement, particle projection and sub-pixel processing (see Figure 1).

3.1 Data arrangement

Modern GPU holds its own memory for fast data fetching locally. Accessing data via GPU memory would be much faster than accessing main memory. Here we distinguish them in terms of host (CPU) memory and device (GPU) memory. Original CPU based PBVR handles all data in host memory causing great time lost in data fetch among the particles. The bandwidth between the device and the device memory is much higher than the bandwidth between the device and host memory. In our test environment, the host memory bandwidth is about 4GB/s while the GPU memory bandwidth is 64GB/s. Due to over 100MB of data referred in rendering processing, these data transfer via different memory space will be time costing. Therefore our preoccupation is to minimize data transfer between the host and device. In this case, since the GPU parallel processing manipulate particles data with high speed synchronically, we need to prepare the raw data in
Before the GPU start to work, large amounts of particles are generated in the host memory as a prerequisite preparation. We use a particle-generation method based on the Metropolis algorithm [8], which is widely used as an efficient Monte Carlo technique in chemistry and physics. The subsequence step of point projection will use the vertices array of particles, and then sub-pixel processing step will use the color array of particles. For linking up those two steps, a depth buffer is allocated for detecting the depth order of projected point stage, and an indices buffer is allocated for storing the index of projected point, which will used to referring its color. All these buffers are located in GPU memory for fast fetch.

We use the CUDA built-in function cudaMemcpy to transfer the particles vertices, color and normal information to the device memory in preparation for the particles project stage. Another important work we do in the data arrangement stage in assigning buffer objects and declaring texture reference, which are allocated for the frame buffer writing of final image. CUDA supports a subset of the texturing hardware that the GPU uses for graphics to access texture memory. Reading data from texture memory can have performance benefits due to its stream fetching feature. As the requirement of OpenGL [10] interoperability of CUDA, a buffer object must be registered before memory data can be mapped [9]. Once it is registered, a buffer object can be read from or written to by kernel s using the device memory address returned by the built-in function cudaMemcpy.

3.2 Particle projection

Particles are dispatched into threads separately, and threads works in parallel batching. When we programmed through CUDA, the GPU is viewed as a compute device capable of executing a very high number of threads in parallel. It operates as a coprocessor to the CPU (host): In other words, data-parallel, compute-intensive portions of applications running on the host are off-loaded onto the device. More precisely, a portion of an application that is executed many times, but independently on different data, can be isolated into a function that is executed on the device as many different threads. To that effect, such a function is compiled to the instruction set of the device and the resulting program, called a kernel, is downloaded to the device.

We assigned the particles processing tasks into threads and blocks batching in a grid range. A thread block is a batch of threads that can cooperate together by efficiently sharing data through some fast shared memory and synchronizing their execution to coordinate memory accesses. There is a limited maximum number of threads that a block can contain. However, blocks of same dimensionality and size that execute the same kernel can be batched together into a grid of blocks, so that the total number of threads that can be launched in a single kernel invocation is much larger. Each thread multiplies the particle’s vertex with a projection matrix to gain 2D coordinates and a depth value. The 2D coordinates are used for referring the depth buffer index, so that we can compare the depth in it.

Here memory writing confliction may easily occurs when several threads happen to gain the same coordinates (that means one single depth index is accessed). To eliminate this risk we pre-compare each threads coordinate value when they reach a synchronized check point. In order word, when every thread in one block finish there tasks, we exam each result which is stored in shared memory. When there are same results detected, we compare their depth and discard the larger one. After this predictive operation, each threads participating memory writing now have its identical coordinate and no confliction would occurs (see Figure 2). Since data in shared memory can be fetched very fast, this confliction checking step does not cost much time.

Figure 2. Eliminate the thread confliction.

3.3 Sub-pixel processing

This is the most suitable stage for SIMD processing due to its dimensionally batching feature. In the present method, we assume that the particles are completely opaque. Thus, neither alpha blending nor visibility ordering is required. In our particle model, we consider three attributes of particles, the shape, the size, and the density. The particle shape is assumed to be a sphere because its projection becomes a view-independent shape, a circle. The size of the sphere is characterized by the radius, which we assumed as the pixel side length divided by an even number so that it facilitates a sub-pixel processing. We call the even number divided by two as a sub-pixel level.

Each thread handles one batch of sub-pixels which divided from an image pixel. The average color of sub-pixels can be calculated and written into final frame...
buffer independently, without any additional data fetching, and that means there is no memory reading or writing confliction (see Figure 3).

In the first stage arrays of particles color and normal are prepared in the device memory and in the second stage an array of particles indices is prepared. Assuming the sub-pixel level is \( n \), one screen pixel is divided into \( n \times n \) parts corresponding to \( n \times n \) particles. Each particle is processed by one thread within one threads block. Here we use the index of this particle to refer its normal value in the normal array, and the attenuation can be deduced through the normal. Then we fetch the RGB color by referring the color array. Therefore a shaded color value can be calculated using the attenuation and color. Finally we average all the \( n \times n \) colors within the sub-pixel range to get the RGB color for this screen pixel (see List 1).

![Figure 3. Sub-pixel processing](image)

**List 1: Kernel code for each thread**

```c
// Each thread calculate one screen pixel
Idx.x = BlockIdx.x * BLOCK_SIZE + ThreadIdx.x;
Idx.y = BlockIdx.y * BLOCK_SIZE + ThreadIdx.y;

// Traverse the \( n \times n \) sub-pixels
for each subpixel {
    subIdx.x = Idx.x * n + subpixel.x;
    subIdx.y = Idx.y * n + subpixel.y;
    // Get the point index through index buffer
    p_index = index_buffer[subIdx.x, subIdx.y];
    // Calculate attenuation using normal array
    attenuate = CalAtt(normal_array[p_index]);
    // Calculate color using color array and attenuate
    RGB = color_array[p_index] * attenuate;
}
```

The buffer object which is assigned in the very beginning is now operated in this phase. Remember we used the `cudaGLMapBufferObject` for fast data fetch through texture mapping. Before the kernel of sub-pixel processing can use a texture reference to read from texture memory, the texture reference need to be bound to a texture using `cudaBindTexture`. After the stage finishes its processing, the texture reference in unbinded.

## 4. EXPERIMENT RESULT

To discuss the evolution of particle-based volume rendering using CUDA, we compare the frame rate with conventional CPU based rendering approach. The target data is a regular volume grid of lobster (120\(\times\)120\(\times\)34). We sample this volume by CPU using Metropolis method. The number of sampled particles ranges from 0.5mega to 3mega, and the sub-pixel level ranges from 2 to 8. Typically as the sub-pixel level reach above 4, rendering effect comes quite satisfied comparing with the ray-casting effect. Therefore our discussion of this experiment focuses in the sub-pixel level of 4 and 6. Spontaneously, number of particles should be generated over 1 mega in order to gain a reasonable visibility. Rendering processing is executed in both GPU and CPU approaches. The GPU approach is based on Nvidia GeForce 8800 GTS with 320MB memory and the CPU approach is based on Intel Core 2 1.86 GHz with 1024MB memory. Here we found the rendering speed is greatly enhanced when two approaches are executed in a same machine (see Figures 4 and 5).

![Figure 4. Performance evolution.](image)
5. CONCLUSION AND FUTURE WORK

New capability of modern GPU is suitable for general purpose computing, this feature provide us great chance to accelerate volume rendering directly. In this paper, we have discussed approaches for developing GPU-accelerated PBVR with CUDA and estimate performance enhancement in comparison with CPU rendering. Due to the C-style program framework of CUDA, no intricate programming technique is needed for this implementation.

As future work, we plan to apply particles generation with CUDA, therefore the whole particle-based volume rendering flow can be transferred into GPU. Since there is no host-device transmission ahead, GPU parallel processing power will be able to exert its maximum performance.

REFERENCES