High-Speed Volume Ray Casting with CUDA

Lukáš Maršálek†
Saarland University

Armin Hauber‡
Saarland University

Philipp Slusallek‡
DFKI Saarbrücken & Saarland University

INTRODUCTION

Volume ray casting experiences a renewed interest in the last decade. Largely due to the graphics hardware, which enabled real-time implementations competitive in speed with slicing.

However these implementations [3, 4] need specialized shader languages and are forced to use graphics APIs. It makes implementation of advanced methods difficult and hinders performance, bending the programming and execution model for something it was not designed to.

In late 2006 a new generation of GPUs has been introduced together with CUDA, C-language API [1]. CUDA exposes the hardware not as a streaming graphics processing pipeline but as a general-purpose highly parallel co-processor.

We aim at evaluating this increased flexibility versus any performance loss or missing low-level access to the hardware. For our study we have chosen basic ray casting using regular sampling with front-to-back traversal, pre-integration and early ray termination.

IMPLEMENTATION AND OPTIMIZATIONS

The entire algorithm, including ray generation is implemented in a single CUDA kernel. Both the volume and pre-integration table are stored in textures, utilizing caches and trilinear interpolation.

A key feature of the G92 GPUs is latency hiding by a large number of threads running on each multiprocessor. To take full advantage of that, we manually reduced live register set from 18 to 12 registers, resulting in 85% occupancy and performance boost of 13%.

This was done first by spilling ray direction and color accumulation buffer to the shared memory. Though this layout causes 3-way bank conflicts, it has proven to be the fastest option, as conflict-free SOA layout requires additional registers. Second, we preferred recomputing common expressions in ray-box intersection to storing them. Finally we reduced threads per block to 320, giving optimal balance of register and shared memory usage.

Kernel, when accessing the device memory, should do so in large chunks and take advantage of requests coalescing. Our only direct device memory access is writing the final pixel values. The kernel runs in full 32-bit floats, but the final pixels store 8 bits per component. Together with proper initial buffer alignment, this setup achieves perfect coalescing.

We have also observed a performance hit, if the image is transferred for display using glDrawPixels command. Using full screen polygon textured from the buffer filled in CUDA kernel results in 15% performance boost, as opposed to the naïve CUDA approach.

CONCLUSION

By creating an optimized CUDA implementation of volume ray casting with pre-integration and early ray termination, we present a proof-of-concept that the enhanced flexibility of programming GPUs in C dialect does not come at a performance hit. Rather it enables low-level access to the hardware, outperforming optimized shader implementations by 12% to 36%, and naïve CUDA implementations by 68% to 114%. It also scales well across different hardware (8600 GTS and 8800 GT), as reducing the processing power by a factor of 3.5, reduces the rendering speed only by a factor of 1.3 to 1.9.

Our future work will concentrate on implementing advanced acceleration techniques and applying the method to new research branches like anthropology and bioinformatics, which is enabled by the additional flexibility and high performance.

REFERENCES

[1] NVidia CUDA 2.0 Beta SDK.