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 receives a renewed interest as graphics hardware enables its realtime implementations. Currently [2, 1, 3], special shader languages and graphics APIs are used, making implementation uncomfortable and difficult. It also hinders performance, as it bends the programming model for something it was not designed to. Recently, new generation of GPUs has been introduced together with CUDA, a C-language API. CUDA exposes the hardware not as a streaming graphics pipeline but as a general highly parallel co-processor. We aim at evaluating this new increased flexibility versus any performance losses. For the study we have chosen ray casting with front-to-back traversal, pre-integration, and early ray termination.

Implementation

```c
__global__ void kernel (unsigned int * pixels) {
    __shared__ float3 s_rayDir[320];
    __shared__ float3 s_dst[320];

    int sIndex = tIdx.y * blockDim.x + tIdx.x;
    s_rayDir[sIndex] = ComputeRayDirection();
    IntersectBoundingBox();
    if( (tOut - tIn) > 0.0f ) {
        f = c_rayOrigin + tIn * s_rayDir[sIndex];
        old = tex3D(datasetTex, f);
        while(tIn < tOut) {
            tIn += c_rayStepSize;
            f = c_rayOrigin + tIn * s_rayDir[sIndex];
            next = tex3D(datasetTex, f);
            f = tex2D(preintTexture2D, old, next);
            old = next;
            AccumulateColor(s_dst[sIndex], f);
        }
        pixels[i] = rgbaFloatToInt(s_dst[sIndex]);
    }
}
```

Conflict-free SM usage

Storing float3 variables in block-sized array enables completely conflict-free shared memory access. This enables maximum parallelism in thread execution, servicing 16 requests to the shared memory simultaneously.

Performance results

<table>
<thead>
<tr>
<th></th>
<th>Neghip (64³)</th>
<th>Foot (256³)</th>
<th>VisFemSlice (512³)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRM</td>
<td>46.7</td>
<td>15.2</td>
<td>11.6</td>
</tr>
<tr>
<td>ST</td>
<td>41</td>
<td>13.5</td>
<td>N/A</td>
</tr>
</tbody>
</table>

FPS in 1024x1024 for our (CRM) and Stegmaier [3] (ST) ray caster

Textures can also be utilized in CUDA

We also use efficient texturing capabilities known from graphics APIs, taking advantage of the automatic interpolation and read-only caching for effective access to volume dataset and pre-integration lookup table.

Coalesced global memory access

To make the access to the global memory effective, it is imperative to enable the hardware to group individual requests. This is done through coalescing the reads and writes so that they access linear memory in a well-defined stride.

Conclusion

Our optimized CUDA ray caster presents a proof of concept that the flexibility of programming GPUs in C dialect does not come at a performance hit. Rather it enables low-level optimizations, outperforming optimized shader implementations. Our future work will concentrate on advanced acceleration techniques and applications to new branches like anthropology and bioinformatics.

References


Acknowledgements

This work was supported by International Max Planck Research School for Computer Science in Saarbrücken, Computer Science Dept. of Saarland University and GAUK grant no. 88008.