AN ADVANCED VOLUME RAYCASTING TECHNIQUE USING GPU STREAM PROCESSING

Jörg Mensmann, Timo Ropinski and Klaus Hinrichs
Visualization and Computer Graphics Research Group (VisCG), University of Münster, Germany

Keywords: Direct volume rendering, Raycasting, Stream processing, CUDA.

Abstract: GPU-based raycasting is the state-of-the-art rendering technique for interactive volume visualization. The ray traversal is usually implemented in a fragment shader, utilizing the hardware in a way that was not originally intended. New programming interfaces for stream processing, such as CUDA, support a more general programming model and the use of additional device features, which are not accessible through traditional shader programming. In this paper we propose a slab-based raycasting technique that is modeled specifically to use these features to accelerate volume rendering. This technique is based on experience gained from comparing fragment shader implementations of basic raycasting to implementations directly translated to CUDA kernels. The comparison covers direct volume rendering with a variety of optional features, e. g., gradient and lighting calculations. Our findings are supported by benchmarks of typical volume visualization scenarios. We conclude that new stream processing models can only gain a small performance advantage when directly porting the basic raycasting algorithm. However, they can be advantageous through novel acceleration methods which use the hardware features not available to shader implementations.

1 INTRODUCTION

Raycasting is advantageous compared to other interactive volume visualization techniques due to its high image quality, inherent flexibility, and simple implementation on programmable GPUs. Implementations usually apply general-purpose GPU programming techniques (GPGPU), which skip most of the geometry functionality of the hardware and use fragment shaders to perform raycasting through the volume data set. Modern GPUs support stream processing as an alternative programming model to classical graphics APIs such as OpenGL. These stream processing models, e. g., NVIDIA's CUDA or OpenCL, give a more general access to the hardware and also support certain hardware features not available via graphics APIs, such as on-chip shared memory.

Rendering approaches for volumetric data can be classified as object-order and image-order traversal techniques. Object-order techniques like slice rendering simplify parallelization by accessing the volume data in a regular manner, but cannot easily generate high quality images and are rather inflexible with regard to acceleration techniques. Image-order techniques such as raycasting (Levoy, 1990), on the other hand, can generate good visual results and can be easily accelerated, e. g., with early ray termination (ERT) or empty space skipping. However, the ray traversal through the volume leads to highly irregular memory access. This can undermine caching and also complicate efforts towards a parallel implementation. Volume raycasting implemented as a fragment shader can give interactive results for reasonably sized data sets, even with on-the-fly gradient calculation and local illumination. More advanced techniques such as gradient filtering or ambient occlusion are still problematic because of the large number of volume texture fetches. In contrast to many other applications of stream processing that often reach high speedup factors, volume raycasting already uses the graphics hardware instead of the CPU. Hence, no major speedups are expected simply by porting a raycasting shader to CUDA. However, fragment shader implementations do not allow sharing of data or intermediate results between different threads, i. e., rays, which therefore have to be fetched or recalculated over and over again. More general programming models exploiting fast on-chip memory could allow a massive reduction in the number of memory transactions and therefore make such advanced visualization techniques available for interactive use.

Mensmann J., Ropinski T. and Hinrichs K. (2010).
AN ADVANCED VOLUME RAYCASTING TECHNIQUE USING GPU STREAM PROCESSING.
In *Proceedings of the International Conference on Computer Graphics Theory and Applications*, pages 190-198
DOI: 10.5220/0002843201900198

In this paper we first examine the general suitability of stream processing for direct volume rendering (DVR) by comparing CUDA- and shader-based raycasting implementations. Afterwards, we discuss acceleration techniques that utilize the additional device features accessible through CUDA and introduce a novel slab-based approach, going beyond what is possible with shader programming.

2 RELATED WORK

GPU-based Volume Raycasting. GPU-based volume raycasting techniques were first published by (Röttger et al., 2003) and by (Krüger and Westermann, 2003). These approaches use a proxy geometry, most often resembling the data set bounding box, to specify ray parameters, either through an analytical approach or by rendering the proxy geometry into a texture.

To speed up rendering, or to support data sets not fitting in GPU memory, the volume can be subdivided into bricks (Scharsach et al., 2006) through which rays are cast independently, while compositing the results afterwards. (Law and Yagel, 1996) presented a bricked volume layout for distributed parallel processing systems that minimizes cache thrashing by preventing multiple transfer of the same volume data to the same processor in order to improve rendering performance. (Grimm et al., 2004) used this approach to get optimal cache coherence on a single processor with hyper-threading.

GPU Stream Processing. New programming interfaces for stream processing allow to bypass the graphics pipeline and directly use the GPU as a massively parallel computing platform. The stream processing model is limited in functionality compared to, e. g., multi-CPU systems, but can be mapped very efficiently to the hardware.

NVIDIA introduced CUDA as a parallel computing architecture and programming model for their GPUs (Nickolls et al., 2008). AMD/ATI support similar functionality through their Stream SDK (AMD, 2009). To have a vendor-neutral solution, OpenCL was developed as an industry standard (Munshi, 2008), but at the time of writing this paper stable implementations were not yet publicly available. For this paper, we have chosen CUDA as a platform for evaluating raycasting techniques because it is the most flexible of the current programming models, it is available for multiple operating systems, and it shares many similarities with the OpenCL programming model.

Besides for numerical computations, CUDA has been used for some rendering techniques, including raytracing (Luebke and Parker, 2008). A simple volume raycasting example is included in the CUDA SDK. (Maršálek et al., 2008) also demonstrated proof of concept of a simple CUDA raycaster and did a performance comparison to a shader implementation. Their results showed a slight performance advantage for the CUDA implementation, but they did not incorporate lighting or other advanced rendering techniques. (Kim, 2008) implemented bricked raycasting on CUDA, distributing some of the data management work to the CPU. He focused on streaming volume data not fitting in GPU memory and did not use all available hardware features for optimization, such as texture filtering hardware. (Smelyanskiy et al., 2009) compared raycasting implementations for Intel's upcoming Larrabee architecture and CUDA, focusing on volume compression and not using texture filtering. (Kainz et al., 2009) recently introduced a new approach for raycasting multiple volume data sets using CUDA. It is based on implementing rasterization of the proxy geometry with CUDA instead of relying on the usual graphics pipeline.

3 RAYCASTING WITH CUDA

3.1 CUDA Architecture

While using the same hardware as shader programs, CUDA makes available certain features that are not accessible by applications through graphics APIs. In contrast to shader programs, a CUDA kernel can read and write arbitrary positions in GPU *global memory*. To achieve maximum bandwidth from global memory, the right access patterns have to be chosen to *coalesce* simultaneous memory accesses into a single memory transaction.

Each multiprocessor on a CUDA device contains a small amount of on-chip memory that can be accessed by all threads in a thread block and can be as fast as a hardware register. This *shared memory* is not available to shader programs. The total amount of shared memory in each multiprocessor—and therefore the maximum amount available to each thread block—is limited to 16 kB with current hardware.

The size and distribution of CUDA thread blocks must be controlled manually. The block size is limited by the available hardware registers and shared memory. Each thread block can use a maximum of 16,384 registers, distributed over all its threads. With a block size of 16×16 this would allow 64 registers per thread, while with a smaller block size of 8×8 the

available registers increase to 256. At most half of these should be used per block to allow running multiple thread blocks on a multiprocessor at the same time. This means that a complex kernel must be run with a smaller block size than a simple one. A similar restriction applies to the use of shared memory.

3.2 Accelerating Raycasting

While easy to implement, the basic raycasting algorithm leaves room for optimization. Many techniques have been proposed for DVR, from skipping over known empty voxels (Levoy, 1990) to adaptively changing the sampling rate (Röttger et al., 2003). Most of these techniques are also applicable to a CUDA implementation. In this paper, we rather focus on techniques that can use the additional capabilities of CUDA to get a performance advantage over a shader implementation.

Many volume visualization techniques take a voxel's neighborhood into account for calculating its visual characteristics, starting with linear interpolation, to gradient calculations of differing complexity, to techniques for ambient occlusion. As the neighborhoods of the voxels sampled by adjacent rays do overlap, many voxels are fetched multiple times, thus wasting memory bandwidth. Moving entire parts of the volume into a fast cache memory could remove much of the superfluous memory transfers.

As noted in Section 3.1, each multiprocessor has available 16 kB of shared memory, but less than half of this should be used by each thread block to get optimal performance. Using the memory for caching of volume data would allow for a subvolume of 16^3 voxels with 16 bit intensity values. While accessing volume data cached in shared memory is faster than transferring them from global memory, it has some disadvantages compared to using the texturing hardware. First, the texturing hardware directly supports trilinear filtering, which would have to be performed manually with multiple shared memory accesses. Second, the texturing hardware automatically handles out-of-range texture coordinates by clamping or wrapping, and removes the need for costly addressing and range checking. Finally, the texture hardware caching can give results similar to shared memory, as long as the access pattern exhibits enough locality.

When a volume is divided into subvolumes that are moved into cache memory, accessing neighboring voxels becomes a problem. Many per-voxel operations like filtering or gradient calculation require access to neighboring voxels. For voxels on the border of the subvolumes much of their neighborhood is not directly accessible any more, since the surrounding voxels are not included in the cache. They can either be accessed directly through global memory, or included into the subvolume as border voxels, thus reducing the usable size of the subvolume cache. Moving border voxels into the cache reduces the usable subvolume size to 14^3, with 33% of the cache occupied with border data. Hence this would substantially reduce the efficiency of the subvolume cache.

Bricking implementations for shader-based volume raycasting often split the proxy geometry into many smaller bricks corresponding to the subvolumes and render them in front-to-back order. This requires special border handling inside the subvolumes and can introduce overhead due to the multitude of shader calls. A CUDA kernel would have to use a less flexible analytical approach for ray setup, instead of utilizing the rasterization hardware as proposed by (Krüger and Westermann, 2003), or implement its own rasterization method (Kainz et al., 2009). As described above, due to the scarce amount of shared memory, the total number of bricks would also be quite high, increasing the overhead for management of bricks and compositing of intermediate results. The bricking technique described in (Law and Yagel, 1996) is specially designed for orthographic projection, for which the depth-sorting of the bricks can be simplified significantly, compared to the case of perspective projection. Their technique also relies on per-brick lists, where rays are added after they first hit the brick and removed after leaving it. This list handling can be efficiently implemented on the CPU, but such data structures do not map efficiently to the GPU hardware. (Kim, 2008) works around this problem by handling the data structures on the CPU. As his aim is streaming of data not fitting into GPU memory, the additional overhead is of no concern, in contrast to when looking for a general approach for volume rendering.

To summarize, a direct bricking implementation in CUDA is problematic because only a small amount of shared memory is available and the ray setup for individual bricks is difficult. Therefore we will introduce an acceleration technique which is better adapted to the features and limitations of the CUDA architecture in Section 5.

4 BASIC RAYCASTING

As illustrated in Figure 1, the basic raycasting algorithm can be divided into three parts: initialization and ray setup, ray traversal, and writing the results. A fragment shader implementation uses texture fetches for retrieving ray parameters, applying transfer functions, and for volume sampling, utilizing the texturing

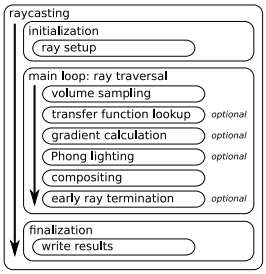

Figure 1: Building blocks for raycasting algorithms.

hardware to get linear filtering.

In a CUDA implementation using textures for the ray start and end points does not have an advantage over memory reads, as no filtering is necessary and coalescing can be achieved easily. Performance differences are more important inside the raycasting loop. Both voxel sampling and transfer function lookup require filtering, so using textures is the natural choice. Our implementation first renders the proxy geometry into OpenGL textures to get the ray start and end points, which can be accessed as CUDA buffer objects through global memory.

The raycasting kernel is then started with the chosen thread block size, with each thread in the block corresponding to a single ray. Following the scheme illustrated in Figure 1, the kernel first performs ray setup using the ray parameter buffers before entering the main loop. Inside the loop the texture fetches are performed and lighting calculation is applied before compositing the intermediate result and advancing the current position on the ray. When the end of a ray is reached the fragment color is written to an output buffer. It is copied to the screen when processing of all thread blocks has completed.

If early ray termination is active, the main loop is terminated before reaching the ray end when the compositing results in an alpha value above a certain threshold. Since all threads in a warp operate in lock step, the thread has to wait for all the other rays to terminate by either reaching their end or through ERT. This is a hardware limitation, hence it also applies to the fragment shader implementation. In practice, however, this is of no concern, as neighboring rays usually exhibit a coherent behavior with regard to ray length and ERT.

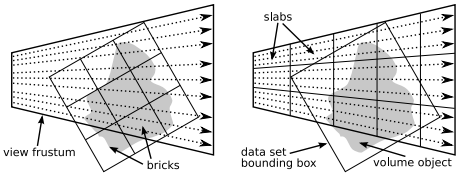

Figure 2: Bricking (object-order) and slab-based (image-order) approach for volume raycasting.

5 SLAB-BASED RAYCASTING

5.1 Approach

Since the bricking described in Section 3.2 is an object-order technique that is not well suited for a CUDA implementation, we introduce an alternative caching mechanism that can be used in image-order by dividing the volume into *slabs*. In contrast to bricking, rays instead of voxels are grouped to build a slab. The screen is subdivided into rectangular regions and stacked slabs reaching into the scene are created, as shown in Figure 2. While for orthogonal projection the structure of a slab is a simple cuboid, it has the form of a frustum for perspective projection.

It would be optimal to move all voxels contained in a slab into shared memory. But unlike bricks, slabs are neither axis-aligned in texture space nor do they have a simple cuboid structure. Therefore either a costly addressing scheme would be required, or large amounts of memory would be wasted when caching the smallest axis-aligned cuboid enclosing the slab. As described in Section 3.2, both alternatives are not suitable for a CUDA implementation. However, a more regular structure can be found after voxel sampling. All rays inside a slab have approximately the same length and therefore the same number of sample points. Saving the voxel sampling results for all rays in a slab leads to a three-dimensional array which can easily be stored in shared memory.

Caching these data does not give a performance advantage *per se*, when samples are only accessed once. But several lighting techniques need to access neighborhood voxels regularly, e. g., ambient occlusion or even basic gradient calculation. When these techniques access the same sample position multiple times, memory bandwidth and latency are saved. Unfortunately, the relation between adjacent samples in the cache is somewhat irregular, as rays are not parallel when applying perspective projection, and therefore the distance between sample points differs. However, often not the exact neighborhood of a voxel is needed but an approximation is sufficient. For large viewport resolutions adjacent rays are close to par-

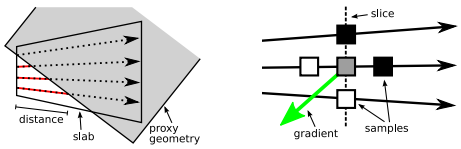

Figure 3: Start point preprocessing (left) and slab-based gradient calculation (right).

allel even with perspective projection, hence for approximation purposes one can consider them as parallel. Gradient calculation can then use the same simple addressing scheme as known from conventional raycasting to access neighboring voxels, although in this case the resulting gradients are relative to the eye coordinate system instead of the object coordinate system. While relying on an explicitly managed cache in shared memory, this method also makes use of the implicit cache of the texturing hardware when sampling the voxels that get written into shared memory. Therefore the two cache levels complement each other.

5.2 CUDA Implementation

Just as with the implementation of the basic raycasting algorithm, also for the slab-based raycasting each thread corresponds to a ray and ray setup is performed through the ray parameter textures. However, the start points must be adapted for the slab structure, as described below. The main loop traverses the rays through the slabs, calculating the gradients using the cache memory. Special handling is necessary for border voxels and for early ray termination.

Start Point Preprocessing. The slab algorithm relies on the fact that voxels are sampled by an advancing ray-front and that sample points which are adjacent in texture space also lie close together in the cache. This only holds true as long as the view plane is parallel to one side of the proxy geometry cube, as otherwise ray start positions have different distances to the camera. This would result in incorrect gradients, since voxels adjacent in the volume may lie on different slices in the slab cache. A solution to the problem is modifying all ray start points in a slab to have the same distance to the camera as the one closest to the camera, as illustrated in Figure 3 (left). We use shared memory and thread synchronization to find the minimum camera distance over all rays in a block and then move the start point of each ray to have this minimum distance to the camera. Moving the start points does not lead to additional texture fetches, as the texture coordinates will lie outside of the interval $[0,1]^3$, which is checked before each 3D texture fetch.

Main Loop. The main rendering loop consists of two parts. In the first part, the slab cache is filled with samples by traversing the ray. As a ray typically creates too many samples to fit in the slab cache completely, the slap depth *sd* controls the number of samples to write into the cache per ray at the same time. Rays with the same distance to the camera lie on the same *slice* in the slab. After thread synchronization the second part of the main loop uses the recently acquired samples to apply lighting and compositing. The ray traversal is started from the beginning for the slab, but now the samples are read directly from shared memory instead of the texture.

Gradient Calculation. A gradient is calculated by taking into account adjacent samples on the same slice from the top, left, bottom, and right rays (as seen from the view point), and the next and previous samples on the current ray, as illustrated for the 2D case in Figure 3 (right). The gradients are therefore calculated in eye space and need to be transformed to object space for the lighting calculation. This results in gradients similar to the default gradient calculation, as shown in Figure 4 (left).

Border Handling. As with bricking, accessing the neighborhood of samples on the border of a slab requires special handling. This is necessary for gradient calculation, because ignoring voxels not accessible through the cache for gradient calculation leads to discontinuities in the gradients, which get visible as a grid pattern in the final image (Figure 4 right). Directly accessing surrounding voxels would require retrieving additional ray parameters for rays outside the slab to calculate the relevant voxel positions. Hence, including the voxels into the cache is more reasonable, even if this reduces the usable size. To include surrounding voxels, we add all rays adjacent to the slab. For these border rays only the first part of the main loop needs to be executed, to write the corresponding samples into the cache for access by the gradient calculation of the inner rays in the second part.

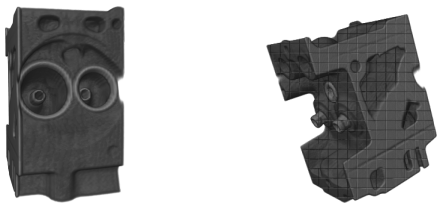

Figure 4: (left) Phong shading applied to the engine data set with gradient calculation. (right) The grid pattern of the thread blocks becomes visible through incorrect gradients when border handling is not performed.

Early Ray Termination. As data sampled by one ray is also used for the gradient calculation in adjacent rays, early ray termination can not stop the traversal of a single ray without taking its neighbors into account. Therefore it must be determined if all rays in a slab have reached the required alpha threshold before terminating further ray traversal. The necessary synchronization can be easily performed using a flag in shared memory.

6 RESULTS

6.1 Testing Methodology

To get meaningful performance data for comparing CUDA and fragment shader raycasting, we have implemented feature-identical versions of a raycaster for both cases, using CUDA version 2.1 and OpenGL shaders implemented in GLSL, running on Linux. The raycasters are integrated into the *Voreen* volume rendering framework (Meyer-Spradow et al., 2009) and use the proxy geometry and corresponding ray start and end positions generated using OpenGL. To get comparable results, our measurements were confined to the actual fragment shader or kernel call, not counting time for rendering the proxy geometry or converting textures from OpenGL to CUDA format. The CUDA kernels were timed using the asynchronous event mechanism from the CUDA API, while for shader raycasting a high-precision timer was used, enclosing the shader execution with calls of `glFinish()` to ensure correct results. Each volume object was rotated around the Y-axis, while measuring the average frame rate over 100 frames. The tests were conducted on two different systems, one equipped with an Intel Core 2 Duo E6300 CPU and an NVIDIA GeForce 8800 GT, the other with a Core 2 Quad Q9550 and a GeForce GTX 280.

The tests start with the simple raycasting algorithm (*RC*), before adding a transfer function (*TF*) and Phong lighting with on-the-fly gradient calculation using central differences and early ray termination (*PH*), and finally performing an expensive gradient filtering (*GF*). Advanced techniques include all previous features, e. g., Phong lighting includes a transfer function. Table 1 lists the number of texture fetches per sample point for the individual techniques. We have tested our implementations with several data sets and chose two representative volumes of different sizes and with different transfer functions for comparing the different techniques. The *engine* is dense with few transparency, while the larger *vmhead* is semi-transparent. Renderings of the data sets with the different techniques are shown in Figure 5.

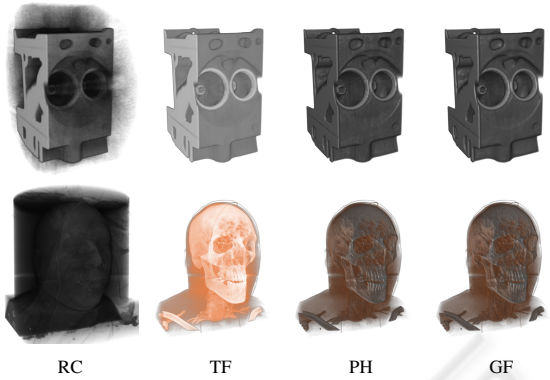

Figure 5: Results of rendering the *engine* and *vmhead* data sets with different raycasting techniques.

Table 1: Register usage and number of texture fetches per sample point for the different raycasting techniques.

technique	regs	fetches
basic raycasting (RC)	15	1
transfer function (TF)	19	2
Phong shading (PH)	33	8
gradient filtering (GF)	57	56

6.2 Basic Raycaster

Table 2 lists frame rates of our raycasting implementations, tested with different GPUs, viewport resolutions, and data sets. It is notable that the GeForce GTX 280 achieves significant speedups for the CUDA implementation for all techniques except *PH*, while with the 8800 GT significant speedups are only found with the *RC* technique and 1024^2 viewport size, the GLSL implementation being close to equal or faster for all other cases. The frame rate differences between GLSL and CUDA reach up to 30%, with one outlier even at +42%. For the 8800 GT increasing the viewport size also increases the speedup, while the speedup for the 280 GTX mostly stays the same. Switching from the *engine* data set to the larger *vmhead* increases the speedup for the 280 GTX, while this is less significant for the 8800 GT.

As the selection of thread block size can have a tremendous influence on the performance of a CUDA kernel, we tested all benchmark configurations with several block sizes to find the optimal block size bs_{opt}. Figure 6 shows the effect of the block size on the frame rate.

While the advanced techniques are more costly since they perform more texture fetches, they also require more hardware registers to run (compare Table 1). Due to the limited availability of hardware registers, this restricts the number of active thread blocks per multiprocessor. The GTX 280 has twice as many

Table 2: Performance results in frames per second for basic raycasting implemented with GLSL and CUDA. The CUDA raycasting was run with different block sizes and results for the optimal block size bs_{opt} are given.

		view-	*engine* data set ($256^2 \times 128$, 8 bit)				*vmhead* data set ($512^2 \times 294$, 16 bit)			
techn.	device	port	GLSL	CUDA	speedup	bs_{opt}	GLSL	CUDA	speedup	bs_{opt}
RC	8800GT	512^2	291.1	300.4	+3.2%	16×8	72.2	64.1	−11.2%	16×20
		1024^2	81.0	96.9	+19.6%	16×32	48.3	56.3	+16.6%	16×28
	GTX280	512^2	380.0	496.0	+30.5%	12×8	121.2	158.5	+30.8%	8×8
		1024^2	124.5	147.8	+18.7%	16×8	70.5	100.2	+42.1%	16×18
TF	8800GT	512^2	194.2	173.5	−10.7%	8×16	68.1	59.4	−12.8%	8×16
		1024^2	61.1	62.3	+2.0%	16×24	38.2	37.4	−2.1%	16×24
	GTX280	512^2	317.4	358.1	+12.8%	16×12	118.0	153.9	+30.4%	8×16
		1024^2	100.9	110.6	+9.6%	8×16	64.6	82.7	+28.0%	16×16
PH	8800GT	512^2	60.2	43.6	−27.6%	8×8	21.5	22.0	+2.3%	8×16
		1024^2	17.1	14.6	−14.6%	16×12	12.0	9.9	−17.5%	16×12
	GTX280	512^2	95.2	77.6	−18.5%	8×8	40.7	38.1	−6.4%	8×16
		1024^2	25.5	22.5	−13.3%	16×8	18.0	17.2	−4.4%	16×8
GF	8800GT	512^2	8.9	6.7	−24.7%	8×16	4.6	3.4	−26.1%	8×16
		1024^2	2.5	2.1	−16.0%	8×16	1.7	1.6	−5.9%	8×16
	GTX280	512^2	9.5	10.4	+9.5%	8×8	4.6	5.6	+21.7%	12×8
		1024^2	2.5	2.9	+16.0%	8×8	1.8	2.3	+27.8%	8×8

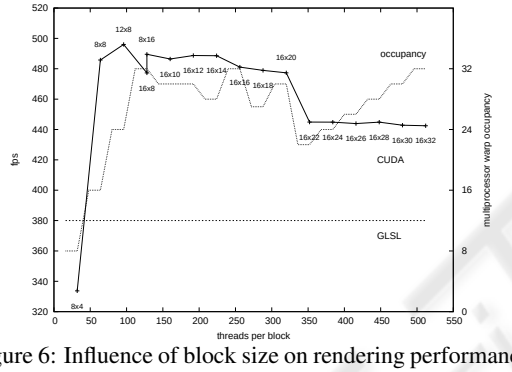

Figure 6: Influence of block size on rendering performance: engine data set rendered using the basic RC technique on a GeForce GTX 280, viewport size is 512^2.

registers available as the 8800 GT and therefore allows larger block sizes for kernels that use many registers. It is notable that for gradient filtering (*GF*), with both a very high register count and a great number of texture fetches, the GTX 280 can again achieve a significant speedup, while it was slower than GLSL for *PH*.

6.3 Slab-based Raycaster

We tested slab-based raycasting on the GTX 280 only, as this GPU proved to be influenced less by high register requirements. The size of the shared memory cache contains $bs_x \times bs_y \times sd$ sampled voxels, depending on the thread block size bs and the slab depth sd. The optimal slab depth sd_{opt} depends on the data set, just as the block size. Results of the slab-based raycaster are presented in Table 3. We added an intermediate viewport size of 768^2 to better analyze the connection between viewport size and speedup factor.

For the basic *RC* technique each sampled voxel is only accessed once, hence caching the slabs cannot improve performance. However, this allows us to measure the overhead for managing the shared memory cache and for fetching additional border voxels. For the tested configurations the overhead is between 23 and 67%. When applying Phong lighting, volume data is accessed multiple times by the gradient calculation, and the slab caching can result in a speedup compared to the basic CUDA raycasting. A performance increase between 12 and 78% is only found with the large 16-bit *vmhead* data set, presumably since the hardware texture cache is less efficient with larger volumes, as for the 8-bit *engine* data set a slight performance decrease is measured. Another reason might be that the early ray termination is less efficient with the slab approach, as only complete slabs can be terminated, not individual rays. The *engine* is solid so the rays are terminated much earlier than with the semi-transparent *vmhead*. It is notable that the speedup decreases with increasing viewport size. When the viewport gets larger, adjacent rays more often hit the same voxels. Hence, there is more locality of texture fetches resulting in more hits in the hardware texture cache. The slab cache is most efficient in the opposite case, when the data set resolution is high compared to the viewport size.

The amount of shared memory required by the raycasting kernels depends on block size and slap depth. For *vmhead* the optimal configuration results in 15,360 bytes which is close to the maximum of 16 kB, hence only one thread block can be active per multiprocessor. Although *engine* only uses up to 7,936 bytes, this does not result in more concurrent thread blocks because of the high register require-

Table 3: Performance results for the CUDA implementation of slab-based raycasting on a GeForce GTX 280. Note that the RC technique is only used to measure the overhead of the slab-based approach.

			engine data set ($256^2 \times 128$, 8 bit)					*vmhead* data set ($512^2 \times 294$, 16 bit)				
technique	regs	viewport	basic	slab	speedup	bs_{opt}	sd_{opt}	basic	slab	speedup	bs_{opt}	sd_{opt}
RC	22	512^2	496.0	186.4	−62.4%	8×16	31	158.5	122.0	−23.0%	16×14	16
		768^2	251.6	85.7	−65.9%	16×16	31	131.1	74.1	−43.5%	16×14	16
		1024^2	147.8	49.2	−66.7%	8×16	31	100.2	43.4	−56.7%	16×30	16
PH	34	512^2	77.6	77.1	−0.6%	8×16	31	38.1	67.9	+78.2%	16×30	16
		768^2	36.2	35.2	−2.8%	16×16	31	27.1	34.1	+25.8%	16×30	16
		1024^2	22.5	19.6	−12.9%	8×16	31	17.2	19.5	+12.7%	16×30	16

ments. with a smaller block size which would allow multiple concurrent thread blocks. This shows that the stream processing approach is not fully effective in hiding the latency of the large number of texture fetches performed by the raycasting algorithm.

6.4 Discussion

The number of required registers seems to be a major factor influencing kernel performance compared to a feature-equivalent shader implementation. Since shaders also benefit from the double bandwidth and twice the number of scalar processors of the GTX 280 compared to the 8800 GT, we suspect that the reason for the greater speedups with the GTX 280 is its support for more hardware registers. It seems that our CUDA implementation is less efficient in utilizing hardware registers than shaders are, therefore profiting more when more registers are available.

The slab-based raycasting can increase rendering efficiency when the same volume data is accessed multiple times, e. g., for gradient calculation. However, it should be noted that the algorithm can be compared to the basic raycasting only to a certain extent, as the gradients are less exact for the slab data. Nonetheless, the results show how much of a difference the use of shared memory can make. We demonstrated that the method is most efficient for high resolution data sets. This is advantageous for typical applications of volume rendering, e. g., medical imaging, where data sets typically have a much higher resolution than *engine*. The algorithm is also more efficient with semi-transparent than with non-transparent data. For data with no transparency this is no real problem as well, as in this case also simpler techniques such as isosurface rendering could be used. Our slab-based method is designed for use with direct volume rendering, which is most useful for semi-transparent data.

7 CONCLUSIONS

We have demonstrated that the CUDA programming model is suitable for volume raycasting and that a CUDA-based raycaster—while not a "silver bullet"—can be more efficient than a shader-based implementation. Factors influencing the speedup are the type of GPU, thread block size, and data set size. We have also shown that using shared memory can bring a substantial performance increase when the same volume data is accessed multiple times. However, hardware restrictions need to be taken into account, as managing the shared memory and especially handling border voxels can introduce a significant overhead. Other factors besides rendering performance should be taken into account as well when choosing a programming model for a raycasting application. A shader implementation supports a wider range of graphics hardware, without depending on a single vendor. Also the integration into existing volume rendering frameworks is easier, e. g., by being able to directly use 2D and 3D textures and render targets from OpenGL. Many of these issues will hopefully be removed by implementations of the OpenCL standard, which is vendor-neutral and supports closer coupling with OpenGL.

As future work it should be investigated whether more complex visualization techniques, such as ambient occlusion, can benefit from the additional hardware resources accessible through stream processing APIs by applying a slab-based approach. As the currently available on-chip memory is a scarce resource, particularly for storing volume data, volume rendering would especially benefit from improvements in this area, which are expected for future hardware.

ACKNOWLEDGEMENTS

The work presented in this publication was partly supported by grants from Deutsche Forschungsgemeinschaft, SFB 656 MoBil (project Z1). The presented

concepts have been integrated into the Voreen volume rendering engine (`http://www.voreen.org`).

REFERENCES

AMD (2009). *Stream Computing User Guide, 1.4-beta*.

Grimm, S., Bruckner, S., Kanitsar, A., and Gröller, M. E. (2004). A refined data addressing and processing scheme to accelerate volume raycasting. *Computers & Graphics*, 28(5):719–729.

Kainz, B., Grabner, M., Bornik, A., Hauswiesner, S., Muehl, J., and Schmalstieg, D. (2009). Ray casting of multiple volumetric datasets with polyhedral boundaries on manycore GPUs. *ACM Transactions on Graphics*, 28(5):1–9.

Kim, J. (2008). *Efficient Rendering of Large 3-D and 4-D Scalar Fields*. PhD thesis, University of Maryland, College Park.

Krüger, J. and Westermann, R. (2003). Acceleration techniques for GPU-based volume rendering. In *Proceedings of IEEE Visualization*, pages 287–292.

Law, A. and Yagel, R. (1996). Multi-frame thrashless ray casting with advancing ray-front. In *Proceedings of Graphics Interfaces*, pages 70–77.

Levoy, M. (1990). Efficient ray tracing of volume data. *ACM Transactions on Graphics*, 9(3):245–261.

Luebke, D. and Parker, S. (2008). Interactive ray tracing with CUDA. Presentation at NVISION conference.

Maršálek, L., Hauber, A., and Slusallek, P. (2008). High-speed volume ray casting with CUDA. *IEEE Symposium on Interactive Ray Tracing*, page 185.

Meyer-Spradow, J., Ropinski, T., Mensmann, J., and Hinrichs, K. (2009). Voreen: A rapid-prototyping environment for ray-casting-based volume visualizations. *IEEE Comp. Graphics and Applications*, 29(6):6–13.

Munshi, A., editor (2008). *The OpenCL Specification, Version 1.0*. Khronos OpenCL Working Group.

Nickolls, J., Buck, I., Garland, M., and Skadron, K. (2008). Scalable parallel programming with CUDA. *ACM Queue*, 6(2):40–53.

Röttger, S., Guthe, S., Weiskopf, D., Ertl, T., and Straßer, W. (2003). Smart hardware-accelerated volume rendering. In *VISSYM '03: Proceedings of the Symposium on Data Visualisation*, pages 231–238.

Scharsach, H., Hadwiger, M., Neubauer, A., Wolfsberger, S., and Bühler, K. (2006). Perspective isosurface and direct volume rendering for virtual endoscopy applications. In *Eurographics/IEEE VGTC Symposium on Visualization*, pages 315–322.

Smelyanskiy, M., Holmes, D., Chhugani, J., Larson, A., et al. (2009). Mapping high-fidelity volume rendering for medical imaging to CPU, GPU and many-core architectures. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1563–1570.