Comparison of High-Speed Ray Casting on GPU using CUDA and OpenGL

Andreas Weinlich, Benjamin Keck, Holger Scherl, Markus Kowarschik and Joachim Hornegger

***Abstract*—Iterative 3D volume reconstruction is one of the most compute- and memory-intensive applications in the field of medical image processing. The iterative reconstruction consists of two major compute intensive steps: Forward- and back-projection. Both steps have to be applied repeatedly in each iteration and several iterations are necessary until a reconstruction result with high image quality is available. As a consequence iterative reconstruction techniques are rarely used in practical CT-like systems. To step towards clinical usage it is mandatory to apply highly parallelized low-cost processing architectures such as the stream processors on current GPUs (Graphics Processing Units). In order to achieve high image quality we implemented the forward-projection using a volumetric ray cast method. We have carefully adapted our implementation to two recent GPU-programming tools, CUDA (NVIDIA Compute Unified Device Architecture) and OpenGL (Open Graphics Language). In terms of execution performance and implementation complexity we compared both tools for the forward-projection step.**

***Index Terms*—computed tomography, iterative reconstruction, volumetric ray casting, CUDA, OpenGL, forward-projection**

I. Introduction

For the last years mostly analytical methods like the filtered back-projection have been used in clinical Cone-beam CT (Computed Tomography) systems in order to achieve 3D volume reconstructions out of acquired 2D projection images. Iterative 3D reconstruction algorithms like SART (Simultaneous Algebraic Reconstruction Technique) or SIRT (Simultaneous Iterative Reconstruction Technique) [1] can produce less reconstruction artifacts [2], i.e. reconstructions using a small amount of projections, even though they are much more time consuming than the conventional Feldkamp algorithm [3]. The iterative reconstruction consists of two major compute- and memory-intensive parts: A forward- and a back-projection step. We recently showed a comparison of latest acceleration technologies for the back-projection step [4]. Especially ray-driven implementations of the forward-projection like a volume ray caster, which are used for their superior precision [5], suffer from their computational demand. Also in other application domains ray casting algorithms are extensively used, like in the field of 2D-3D registration [6]. To overcome the limitations and build real time solutions for clinical application, it is necessary to use hardware architectures with massively parallel computation capabilities. Like in similar applications, one of the most appropriate and cost efficient solutions are modern graphics cards [7]. For example, NVIDIA's GeForce 8800 GTX and QuadroFX 5600, which we utilized for our tests, use 128 stream processors in parallel and can additionally benefit from some hardware-accelerated features like texture interpolation. Recently NVIDIA has developed a C-like general purpose API for these GPUs to implement for example parallelized numerical algorithms.

Unfortunately, the first CUDA versions up to 1.1 had still some drawbacks like missing support for 3D textures. This feature was introduced in the recently published major release, CUDA 2.0. But maybe still the compiler is not as sophisticated as in the OpenGL graphics programming language. Furthermore, as a matter of principle, it can only be used on modern NVIDIA graphics cards. On the other hand there exists another very interesting hardware platform for CUDA applications called NVIDIA Tesla. In this paper we compare highly optimized implementations of ray casting using CUDA 1.1, CUDA 2.0 and OpenGL regarding programming techniques, implementation time, and execution performance.

A. Weinlich, B. Keck and J. Hornegger are with the Friedrich-Alexander-University Erlangen-Nuremberg, Department of Computer Science, Chair of Pattern Recognition (LME), Martensstr. 3, D-91058 Erlangen, Germany.

H.Scherl and M. Kowarschik are with Siemens Healthcare, CV, Medical Electronics & Imaging Solutions, P.O.Box 3260, D-91050 Erlangen, Germany.

II. Related Work

In the medical field, perspective projections are often used to simulate and approximate the physical process of X-ray attenuation. Over two decades ago, Joseph [8] introduced an improved algorithm for forward-projecting rays. His algorithm is not as precise as a ray cast based algorithm, but less computationally complex, which was more important at this time. Later Xu et. al. compared popular interpolation and integration methods for use in CT [5] and showed that a ray cast based algorithm is comparable to the other superior methods regarding the root mean square (RMS) error. Because modern GPUs provide hardware-accelerated interpolation, we decided to implement the forward-projection using ray casting.

The iterative reconstruction performance of graphics accelerators has often been evaluated using OpenGL and shading languages [7], [9].

III. Methods

In this section we describe the principle of the forward-projection step. Second, we explain our CUDA-based and OpenGL-based implementations.

A. Forward-projection

We use a volumetric ray casting approach for the forward-projection step. Its basic functionality is shown in Figure 1 and the algorithm is shown in Algorithm 1. To determine the grey level value of a certain pixel on the image plane, a straight

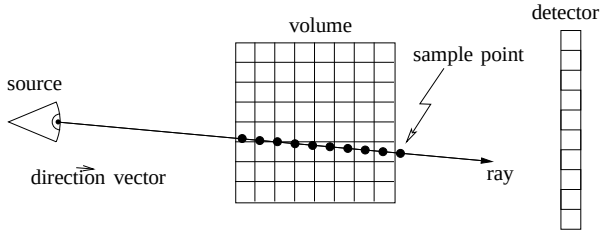

Fig. 1. Ray casting principle.

line ("ray") is drawn pointing from the optical center towards the pixel position. Afterwards voxel intensity values inside the cuboid are sampled equidistantly along the ray. These sampling values add up to the desired gray level value in the image. As a result we get a perspective projection of the volume data.

Algorithm 1 Forward-projection with a ray casting algorithm

```
for all projections do
  compute source position out of projection matrix
  compute inverted projection matrix
  for all rays inside the projection do
    compute ray direction depending on the image plane
    normalize direction vector
    //RAY CASTING
    compute entrance and exit point of the ray to the cuboid
    if ray hits the cuboid then
      set sample point to the entrance point
      initialize the pixel value
      while sample point is inside the cuboid do
        add up the computed sample value at current
        position to the pixel value
        compute new sample point for given step size
      end while
    else
      set pixel value to zero
    end if
    normalize pixel value to world coordinate system units
  end for
end for
```

The physical process of acquiring an X-ray image works just as well. In particular, in this case the optical center depicts the X-ray source whereas the image plane depicts the detector. While Strobel et. al. [10] have shown that the image quality of a reconstruction can be improved by using projection matrices instead of assuming an ideal geometry, we decided to use this parameterization in our implementation.

Furthermore this section describes some general features that are common to both implementations, CUDA as well as OpenGL. There are some different methods to get the direction vector of the ray, which is the first step in the inner for loop in Algorithm 1. A simple one is to take two position vectors, compute the difference vector, and normalize it. Such positions are the optical center, the 3D coordinate of the pixel position, or the points where the ray enters or leaves the cuboid. For example the position of the optical center can be obtained

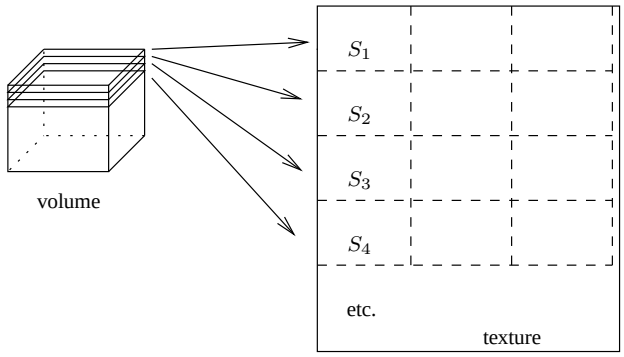

Fig. 2. Volume representation in a 2D texture by Slices S_i.

from the homogeneous projection matrix which is designed to project a 3D point to the image plane. Depending on the output format of the projection (2D image- vs. 3D world-coordinates), this matrix has three or four rows. In the latter case, the vector can be found in the fourth column of the inverted matrix (first three components). In the case of a 3×4 matrix it is possible to drop the fourth column, invert the 3×3 matrix and multiply the inverse with the previously dropped fourth column to get the center position. This holds, because in case of a perspective projection with projection matrices, this fourth column depicts the shift of the optical center to the origin of the coordinate system. But due to the fact that this translation occurs not before the rest of the transformations, these have to be undone in multiplying the inverse. Galigekere et. al. have shown already how to reproject using projection matrices in [11].

In the next step the entrance position of the ray into the volume has to be calculated. The used method to get the entering and leaving points depends on the implementation. Between those points the cube is equidistantly sampled. To get one sampling position, we take the entry vector and add the direction vector multiplied with the step size times a counter variable. The following sampling step itself proves to be crucial for the algorithm's efficiency. In order to get satisfying results, a sub-pixel sampling is required, which introduces a trilinear interpolation.

For a realistic simulation of X-ray imaging, the Beer-Lambert law has to be fulfilled approximately:

$$I = I_0 \cdot e^{-\int_{t(\underline{v}_{\text{source}})}^{t(\underline{v}_{\text{detector}})} \rho(\underline{x}(t))\, dt} \tag{1}$$

The densities p are integrated along the line $\underline{x}(t)$ (or added up in a discrete manner). Afterwards, they are transformed with the exponential-function and multiplied with an initial X-ray intensity to get the target intensity value. This subsequent transformation will not be considered here as it can be computed for example during a post-processing step. For the application in algebraic reconstruction, a pre-processing of the original X-ray images may be also appropriate to fit the ray caster projections.

B. Implementation in CUDA

CUDA offers an easy to use C-like application programming interface with some extensions. There are two different parts in each CUDA implementation: A host part, which executes in a CPU thread, and a device part (kernel), which is invoked by the controlling CPU thread, but runs in parallel on the GPU device. In our case the program instructs the graphics card to create a semi-parallel thread for each ray. On our hardware up to 128 of these threads can be processed in parallel. Most of our CPU code uses CUDA specific API functions for allocating data structures on the device and to transfer data to the graphics memory and back to RAM.

In the kernel code, the inverse of the projection matrix is used to get the ray direction out of the pixel position in the projection image. In order to check whether a sampling position is inside the cuboid, the entrance and exit distances with respect to the optical center are computed. In each step the entrance position is incremented by a step size value until it reaches the exit distance. A critical issue in CUDA 1.1 is the sampling step since it does not provide support for 3D textures. So unfortunately a trilinear hardware interpolation is not available for the CUDA 1.1 API. In consequence, a workaround had to be applied that used just the bilinear interpolation capability of the GPU. It does a successive linear software interpolation in between stacked 2D texture slices (see Figure 2). Therefor, desired values are fetched from proximate stack slices with hardware-accelerated bilinear interpolation. These sampling steps are substituted with only one hardware-accelerated 3D texture fetch in CUDA 2.0 and OpenGL.

C. Implementation in OpenGL

The OpenGL implementation is more tricky in some aspects. This is a consequence of the fact, that OpenGL is intended to be used in graphics applications. Nevertheless there are some similarities like the perspective projection. In the past years, the API itself was made more flexible by means of shader languages, which makes it possible to implement a forward projection using OpenGL [12].

Like in CUDA, the implementation divides into a CPU and a GPU part. The CPU part (OpenGL code) was written in C++. In our implementation the GPU fragment shader program is written in the shader language, GLslang. The OpenGL API invokes this code for each pixel in the projection. Due to the fact that a pixel exactly corresponds to a ray, this threading is the same as in CUDA. However, unlike CUDA, this partitioning can not be defined by the programmer directly. In fact this correspondence is a fixed OpenGL fragment shader feature.

In the OpenGL code, there are some initializations establishing a desktop window for rendering. Furthermore, frame buffer objects are initialized in order to store the projection into a texture. As stated above, the volume data resides in a 3D texture like in CUDA 2.0. This fact allows for the utilization of hardware supported tri-linear interpolation. The projection matrix for an image has to be transformed in order to fit the OpenGL coordinate system. Afterwards some variables are transferred to the shader, the six faces of the cuboid are drawn using vertices, the cuboid is rendered to a texture and finally this texture is copied back to host memory.

During the rendering, the instructions within the shader program are executed instead of the texture lookup. These instructions differ slightly from the corresponding CUDA code. Corners of the 3D texture have been assigned to the corners of the cuboid, so the OpenGL texturing step provides the entrance position of the ray automatically in terms of interpolated texture coordinates. The ray direction vector can be obtained like it was outlined in the last section. In each step the program checks, whether the sampling position is still inside the cuboid. As mentioned, the sampling itself reduces to a simple 3D texture fetch.

	NVIDIA GeForce 8800GTX	NVIDIA QuadroFX 5600
Core clock	575 MHz	600 MHz
Shader clock	1350 MHz	1400 MHz
Memory amount	768 MB	1500 MB
Memory interface	384-bit	384-bit
Memory clock speed	900 MHz	800 MHz
Memory bandwidth	86.4 GB/s	76.8 GB/s

TABLE I
TECHNICAL SPECIFICATION OF BOTH GRAPHICS CARDS USED IN OUR EVALUATION

IV. RESULTS

In order to compare the performance of both approaches, we measured execution times with different test parameters on an NVIDIA GeForce 8800 GTX as well as on an NVIDIA QuadroFX 5600. Even though both graphics cards are assembled with the NVIDIA GPU ”G80” they are slightly different stated in Table I. Our evaluation system is a Fujitsu-Siemens Workstation ”R650” using the Intel 5400 chip set. The graphics cards are connected each via a PCI Express x16 slot.

For measurement purpose we used different projection geometries and volume phantoms. If the phantom fits inside the field of view, there exist rays that do not go through the cuboid at all (case ”far”). These rays consume a minimum of the computation time and the computation finishes noticeably faster compared to the test case where optical center and image plane are close to the cuboid (case ”near”). Associated parameters that have direct impact on the computational complexity of the ray caster are image size (number of pixels and with it number of rays) as well as the sampling rate along one ray (distance of sampling positions compared to the size of a voxel). Due to the fact that in CUDA the execution of the kernel and thus the ordering of the texture fetches can be configured by the block configuration [13], we also compared this parameter for CUDA 1.1 and CUDA 2.0. Large images have some additional side effects. On one hand, they allow a more flexible schedule of threads, on the other hand each ray

Blocksz.	512^2 pixels near	512^2 pixels far	1024^2 pixels near	1024^2 pixels far	2048^2 pixels near	2048^2 pixels far
16×16	48.2	87.7	106	107	409	301
32×8	50.5	101	109	111	412	315
32×16	46.4	113	107	116	411	308
64×4	59.8	127	109	138	424	340
64×8	54.4	129	111	127	415	330
128×2	74.0	132	121	222	425	397
128×4	57.8	124	115	185	431	372
256×1	98.2	140	169	302	449	597
256×2	68.9	124	122	218	448	467
512×1	100	141	167	253	441	593

TABLE II
BLOCK PARAMETER COMPARISON OF RUNTIMES USING CUDA 2.0 ON THE NVIDIA GEFORCE 8800GTX IN SECONDS WITH 400 PROJECTIONS AND DIFFERENT PROJECTION SIZES AT A STEP SIZE OF 0.25 OF THE VOXELSIZE

# Proj.	FoV	512^2 pixels CUDA 1.1	512^2 pixels CUDA 2.0	512^2 pixels OpenGL
1	near	6.22	**1.60**	3.25
	far	6.47	**1.60**	3.24
16	near	14.2	**3.30**	5.32
	far	18.2	**4.97**	6.45
100	near	55.5	**13.1**	21.7
	far	92.5	**24.4**	25.3
400	near	145	**41.8**	47.0
	far	386	**88.7**	90.3

TABLE III
COMPARISON OF RUNTIMES USING THE NVIDIA QUADROFX 5600 IN SECONDS (CUDA 1.1 VS. CUDA 2.0 VS. OPENGL) WITH A PROJECTION SIZE OF 512 SQUARED AND A DIFFERENT NUMBER OF PROJECTIONS AT A STEP SIZE OF 0.25 OF THE VOXELSIZE

needs some initial calculation steps apart from the sampling. Unless otherwise noted, a block consists of 16×16 pixels within the projection. A block parameter comparison for the GeForce 8800 GTX using CUDA 2.0 is shown in Table II. Another important parameter is the number of projections to be acquired from the same volume data. The time required for initialization steps, preparing the data structures and loading the volume data to the device, is spent just once. So, a high number of projections reduces the influence of such preceding computations (e. g. 1.6 seconds for CUDA and 3.2 seconds for OpenGL on the QuadroFX 5600).

If both implementations are well optimized, it is expected that OpenGL will perform better than CUDA 1.1 and comparable to CUDA 2.0.

We use a projection size of 512×512 or 1024×1024 pixels. The resulting execution times for the GeForce 8800 GTX and QuadroFX 5600 using a projection size of 1024×1024 are shown in Table IV and Table V, and for the QuadroFX 5600 in Table III using a projection size of 512×512 and 2048×2048 in Table VI. In Figure IV we give an overview of the dependency on the projection size using the QuadroFX 5600. In order to hit most of the voxels in the volume, the step size (sampling rate) must not be greater than 1 voxel. If we actually do not want to loose information, it should be at most 0.5 of the voxel size. In favor of a smooth projection image a step size of 0.25 voxels would be even better. A direct comparison between GeForce 8800 GTX and QuadroFX 5600 for the computation time depending on the step size is shown in Figure 7. The number of projections that can be computed consecutively depends on the reconstruction algorithm. For example, SART computes only a single projection per volume update. In contrast, SIRT processes all projections consecutively before a volume update is performed in the iteration. Certainly there are algorithms in between such as the ordered subset approach.

In Figure 6 we can see the dependency of the execution time on the chosen step size for most common parameters. The measurements do not include the time required to write-back the projections to the host memory or even to hard disk, because it is not required for a complete GPU implementation of iterative CT reconstruction. Moreover, those times (especially the write back to disk) can be hidden behind the computation of the next slices. For example a projection of 100 images, 1024×1024 including write back takes approximately one additional second on the QuadroFX 5600 (0.35 sec write back to host, 0.19 sec write back to hard disc and 0.54 sec for deletion of data, etc.). In most cases OpenGL and CUDA 2.0 operate two or three times faster than CUDA 1.1. For a small number of projections, the results seem to depend on the other parameters, i.e. the initialization time of the API, which takes longer for OpenGL. In contrast, the tests with 400 projections show a more interesting behavior. The best executed results are highlighted in bold in Table IV, III, V and VI. In Table IV it can be seen that CUDA 2.0 is faster in all tests by a constant offset of approximately 8 seconds on the GeForce 8800 GTX. In Figure the dependency on the step size for the two different geometric setups in a common setting for SIRT (1024×1024 pixels; 400 projections) is shown. The time increases almost linear with the step size except for an offset.

To give an impression of GPUs computational performance we finally compare a specific test case also with a CPU implementation. The CPU implementation is a single-threaded non-optimized straight-forward implementation of the raycast method as stated in Algorithm 1. The program is executed on our test system equipped with two Intel Xeon E5410 processors running at 2.33 GHz. For a simple comparison we used 16 projections 1024×1024 at a step size of 0.25 of the voxel size. Table V proves a performance of 5.16 seconds for such configuration using the "near" field of view setting on the NVIDIA QuadroFX 5600. We measured 764 seconds for the single threaded CPU program. This indicates a maximal speedup factor of 148.

V. DISCUSSION

At higher numbers of projections the execution times for the CUDA implementation which uses 2-D textures to compute

# Proj.	FoV	1024^2 pixels CUDA 1.1	CUDA 2.0	OpenGL
1	near	9.4	**3.8**	12.1
	far	9.4	**3.8**	11.9
16	near	20.6	**7.5**	15.5
	far	27.3	**8.2**	15.5
100	near	86.4	**28.4**	36.4
	far	126	**30.2**	37.3
400	near	299	**107**	115
	far	527	**108**	116

TABLE IV
COMPARISON OF RUNTIMES USING THE NVIDIA GEFORCE 8800GTX IN SECONDS (CUDA 1.1 VS. CUDA 2.0 VS. OPENGL) WITH A PROJECTION SIZE OF 1024 SQUARED AND A DIFFERENT NUMBER OF PROJECTIONS AT A STEP SIZE OF 0.25 OF THE VOXELSIZE

# Proj.	FoV	1024^2 pixels CUDA 1.1	CUDA 2.0	OpenGL
1	near	6.38	**1.60**	3.22
	far	6.71	**1.59**	3.25
16	near	16.2	**5.16**	6.94
	far	21.4	**5.02**	7.09
100	near	70.5	**25.1**	27.4
	far	114	**24.6**	29.5
400	near	245	**99.8**	103
	far	515	**90.9**	109

TABLE V
COMPARISON OF RUNTIMES USING THE NVIDIA QUADROFX 5600 IN SECONDS (CUDA 1.1 VS. CUDA 2.0 VS. OPENGL) WITH A PROJECTION SIZE OF 1024 SQUARED AND A DIFFERENT NUMBER OF PROJECTIONS AT A STEP SIZE OF 0.25 OF THE VOXELSIZE

trilinear interpolations are much longer than for our other implementations using 3D textures (CUDA 2.0 or OpenGL). It is therefore essential to use the hardware-accelerated functions of the GPU in order to optimize the execution performance of our CT reconstruction applications. The constant execution time offset in each test case (approx. 12 seconds in OpenGL and 4 seconds in CUDA 2.0 on the GeForce 8800 GTX) can be explained with the copy process of the volume data to the graphics memory along with some other initializations. With a QuadroFX 5600 card we observed a significantly smaller

# Proj.	FoV	2048^2 pixels CUDA 1.1	CUDA 2.0	OpenGL
1	near	7.70	**1.59**	3.27
	far	7.26	**1.58**	3.26
16	near	37.7	**15.8**	17.7
	far	30.6	**11.4**	13.3
100	near	208	**95.4**	98
	far	173	**67.4**	70.4
400	near	841	**392**	397
	far	864	**284**	290

TABLE VI
COMPARISON OF RUNTIMES USING THE NVIDIA QUADROFX 5600 IN SECONDS (CUDA 1.1 VS. CUDA 2.0 VS. OPENGL) WITH A PROJECTION SIZE OF 2048 SQUARED AND A DIFFERENT NUMBER OF PROJECTIONS AT A STEP SIZE OF 0.25 OF THE VOXELSIZE.

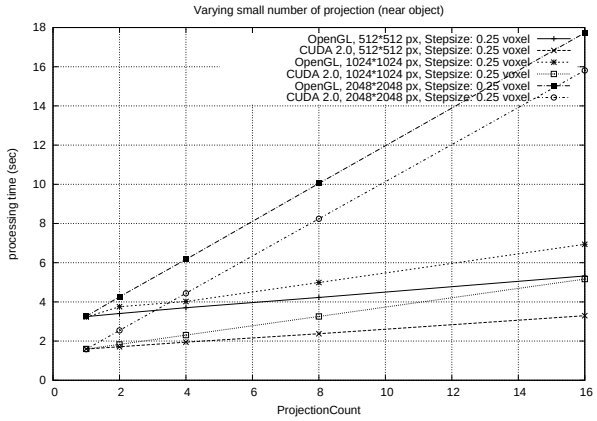

Fig. 3. CUDA 2.0 and OpenGL comparison for varying projection size and a small amount of projections.

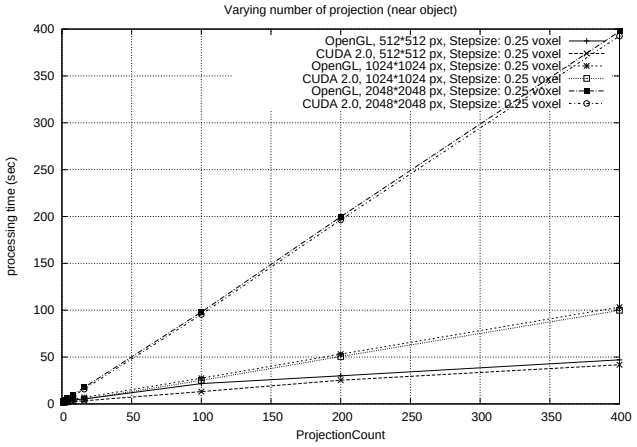

Fig. 4. OpenGL and CUDA 2.0 comparison with similar execution time behavior for varying projection count and size.

difference in initialization time between OpenGL and CUDA. As expected, the increase in runtime is almost linear in the step size and the number of projections. With increasing image sizes, OpenGL and CUDA are able to dispatch the parallel computations more efficiently to the multiprocessors of the GPU up to a certain amount. This is the reason why the execution time increases remarkably slower and does not scale with the number of pixels in an image. Merely at 2048×2048 pixels and an ROI including the complete data, there can be seen a strong increase in execution time. As a consequence, it seems that projection images with 1024×1024 pixels are optimally suited for current GPUs generations. An implementation in OpenGL requires more implementation efforts for non-experts because it was built as a graphics programming language for real-time rendering of vertex-based 3D scenes. In contrast, a ray casting in the C programming language can be more easily ported to CUDA, as it only requires some adaptations for the parallelization strategy. However an OpenGL expert can implement such an algorithm in equivalent time compared to a C-Programmer using CUDA.

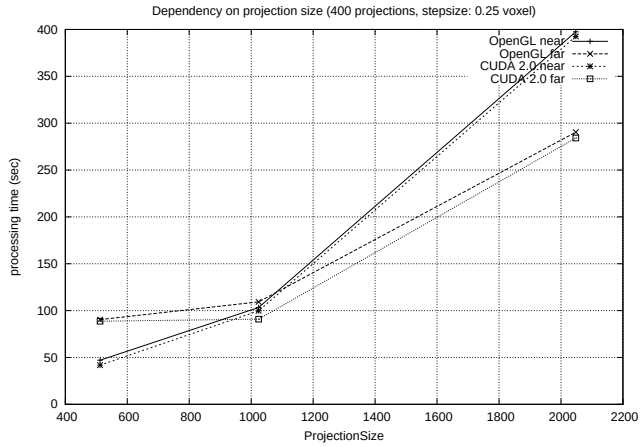

Fig. 5. The projection size dependency on the QuadroFX 5600.

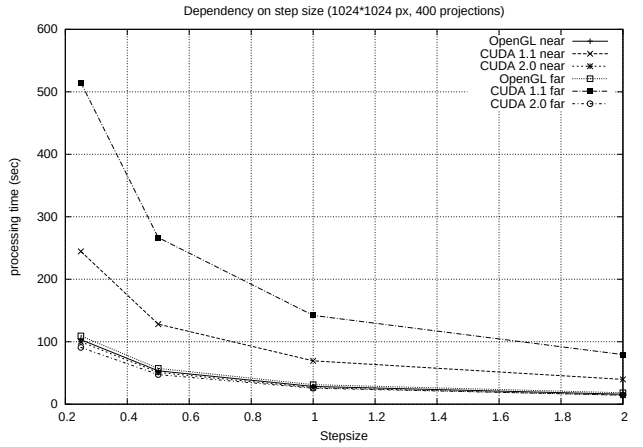

Fig. 6. The stepsize dependency on the QuadroFX 5600.

VI. Conclusion

We have presented three highly optimized implementations of volume ray casting usable i.e. as the forward-projection step in iterative reconstruction. Our comparison of the execution times shows that the performance of the recent CUDA version is even slightly better than an implementation using OpenGL. Older CUDA versions should not be used for ray casting due to the lack of 3D texture support. CUDA unveils the processing power of graphics cards even for programmers that are not specialists in computer graphics. The OpenGL implementation required much more implementation time, however it can also be used with no CUDA capable devices. On the other hand, the Tesla series from NVIDIA can only be used together with CUDA.

Acknowledgments

This work is being supported by Siemens Healthcare, CV, Medical Electronics & Imaging Solutions. We wish to give special thanks to Dr. Klaus Engel who supported us with his wide OpenGL API knowledge.

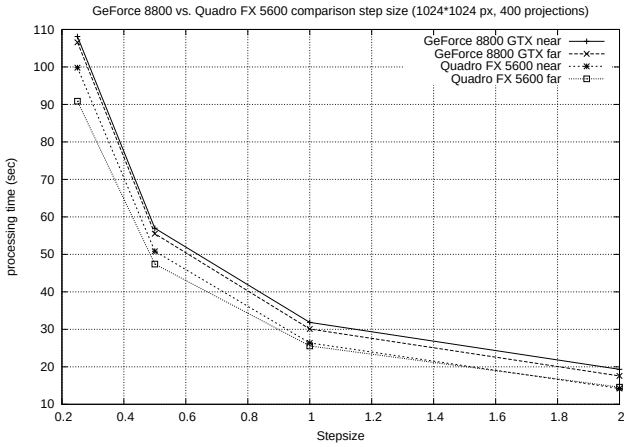

Fig. 7. GeForce 8800 GTX to QuadroFX 5600 comparison on step-size using CUDA 2.0.

References

[1] A. Andersen and A. Kak, "Simultaneous algebraic reconstruction technique (sart): A superior implementation of the art algorithm," *Ultrasonic Imaging*, vol. 6, no. 1, pp. 81–94, January 1984.

[2] K. Mueller and R. Yagel, "Rapid 3d cone-beam reconstruction with the algebraic reconstruction technique (art) by utilizing texture mapping graphics hardware," *Nuclear Science Symposium, 1998. Conference Record. 1998 IEEE*, vol. 3, pp. 1552–1559, 1998.

[3] L. Feldkamp, L. Davis, and J. Kress, "Practical cone-beam algorithm," *Journal of the Optical Society of America*, vol. A1, no. 6, pp. 612–619, 1984.

[4] H. Scherl, B. Keck, M. Kowarschik, and J. Hornegger, "Fast GPU-Based CT Reconstruction using the Common Unified Device Architecture (CUDA)," in *Nuclear Science Symposium, Medical Imaging Conference 2007*, E. C. Frey, Ed., 2007, pp. 4464–4466.

[5] F. Xu and K. Mueller, "A comparative study of popular interpolation and integration methods for use in computed tomography," *Biomedical Imaging: Nano to Macro, 2006. 3rd IEEE International Symposium on*, pp. 1252–1255, April 2006.

[6] A. Kubias, F. Deinzer, T. Feldmann, S. Paulus, D. Paulus, B. Schreiber, and T. Brunner, "2d/3d image registration on the gpu," in *Proceedings of the 7th Open German/Russian Workshop on Pattern Recognition and Image Understanding (OGRW), FGAN-FOM*, Ettlingen, 2007.

[7] K. Mueller, F. Xu, and N. Neophytou, "Why do commodity graphics hardware boards (GPUs) work so well for acceleration of computed tomography?" in *SPIE Electronic Imaging Conference*, San Diego, 2007, (Keynote, Computational Imaging V).

[8] P. M. Joseph, "An improved algorithm for reprojecting rays through pixel images," *IEEE Transactions on Medical Imaging*, vol. MI-1, no. 3, pp. 192–196, 1982.

[9] M. Churchill, "Hardware-accelerated cone-beam reconstruction on a mobile C-arm," in *Proceedings of SPIE*, J. Hsieh and M. Flynn, Eds., vol. 6510, 2007, p. 65105S.

[10] N. K. Strobel, B. Heigl, T. M. Brunner, O. Schuetz, M. M. Mitschke, K. Wiesent, and T. Mertelmeier, "Improving 3D image quality of x-ray C-arm imaging systems by using properly designed pose determination systems for calibrating the projection geometry," in *Medical Imaging 2003: Physics of Medical Imaging. Edited by Yaffe, Martin J.; Antonuk, Larry E. Proceedings of the SPIE, Volume 5030, pp. 943-954 (2003).*, ser. Presented at the Society of Photo-Optical Instrumentation Engineers (SPIE) Conference, M. J. Yaffe and L. E. Antonuk, Eds., vol. 5030, Jun. 2003, pp. 943–954.

[11] D. H. R. Galigekere, K. Wiesent, "Cone-beam reprojection using projection-matrices," *IEEE Transactions on Medical Imaging*, vol. 22, no. 10, pp. 1202–1213, 2003.

[12] K. Müller, "Fast and accurate three-dimensional reconstruction from cone-beam projection data using algebraic methods," Ph.D. dissertation, Department of computer and information science, Ohio State University, Columbus, Ohio, USA, 1998.

[13] N. Corp., "NVIDIA CUDA Compute Unified Device Architecture Programming Guide," 2007. [Online]. Available: http://www.nvidia.com/cuda