# GPU Acceleration for Digitally Reconstructed Radiographs using Bindless Texture Objects and CUDA/OpenGL Interoperability

Marwan Abdellah[1†], *Student Member, IEEE, EMBS*
Ayman Eldeib[2§], *Senior Member, IEEE, EMBS*, and Mohamed I. Owis[3‡], *Member, IEEE, EMBS*

***Abstract*— This paper features an advanced implementation of the X-ray rendering algorithm that harnesses the giant computing power of the current commodity graphics processors to accelerate the generation of high resolution digitally reconstructed radiographs (DRRs). The presented pipeline exploits the latest features of NVIDIA Graphics Processing Unit (GPU) architectures, mainly bindless texture objects and dynamic parallelism. The rendering throughput is substantially improved by exploiting the interoperability mechanisms between CUDA and OpenGL. The benchmarks of our optimized rendering pipeline reflect its capability of generating DRRs with resolutions of $2048^2$ and $4096^2$ at interactive and semi interactive frame-rates using an NVIDIA GeForce 970 GTX device.**

## I. Introduction

The recent advances in medical imaging are correlated with the rapid evolution of cutting edge imaging technologies such as Computed Tomography (CT), Magnetic Resonance Imaging (MRI), Positron Emission Tomography (PET) and other modalities [1]. With the advent of these state-of-the-art imaging scanners, fast and high resolution scans are affordable, allowing excellent imaging of very fine structures of the human tissue. This leap comes with the cost of producing huge volumes of highly dynamic imaging data. Consequently, the active presence of high-end workstations and high performance rendering pipelines is crucial for efficient handling of large imaging artifacts produced by these scanners.

*X-ray volume rendering* is known to be one of the most frequent rendering techniques that is used for visualizing medical volumes generated from imaging scanners [2], [3]. It is of high importance in digital radiography because the physicians are highly trained for exploring and interpreting X-ray radiographs [4]. This technique is employed in image guided surgery to generate DRRs that can be used to align the orientation of the images acquired during an operation with another volume that was acquired a priori [5]. The generation of DRRs is a computationally expensive process that requires an optimized parallel execution engine capable of handling high resolution volume data interactively. Due to the embarrassingly parallel nature of the technique, it can be accelerated on parallel many core architectures, for instance Field Programmable Gate Arrays (FPGAs) and GPUs.

During the last decade, GPUs have evolved from being highly domain-specific engines dedicated for computer graphics developers to extremely powerful and general purpose high performance computing platforms [6]. The rollout of Compute Unified Device Architecture (CUDA) in 2004 introduced another significant leap that has revolutionized GPU programming [7]. It has exposed the underlying architecture of the GPU at no overhead required to get acquainted with its complex pipeline. Compared to the legacy approach of accessing this pipeline through graphics-specific Application Programming Interface (API) such as OpenGL or Direct3D, the flexible API provided with CUDA has made it relatively trivial for scientists and researchers to harness the giant computing power of the GPU to accelerate their parallel algorithms.

CUDA-enabled GPU architectures have rapidly evolved from the Tesla architecture (compute capability 1.0) to the Fermi generation (compute capability 2.0). Fermi-based GPUs have supported double precision, caching mechanisms and concurrent kernel execution. In 2012, the Kepler architecture (compute capability 3.0) was deputed to bring yet another significant milestone in GPU programming. The Kepler generation has introduced several features that enriched the performance and the functionality of the GPU such as *dynamic parallelism* and *bindless texture objects*. These features are further improved in the following Maxwell micro-architecture [8]. Dynamic parallelism increases concurrency and the occupancy of the GPU by reducing the communication between the device and the host and allowing kernel launches within the device. Generally, textures are employed to maximize memory bandwidth and eliminate coherency constrains for applications that require high spatial locality. In addition to preserving all the functional aspects of the old texture references, texture objects have gained performance by creating them during the run time and eliminating the overhead that was associated of binding and unbinding texture references. This permits strong-scaling performance of multi-GPU applications. Moreover, the texture objects count is not limited by the hardware restrictions as before. This advantage can scale applications that have high texture requirements on a single GPU [8]. Designing high performance rendering applications requires splitting their pipeline between CUDA for executing data-parallel compute kernels and OpenGL for processing geometry-based operations. The optimization of the communication mecha-

[1]Marwan Abdellah is a M.Sc graduate from the Department of Biomedical Engineering, Faculty of Engineering, Cairo University, Egypt.
[†] Corresponding author, Email: *marwan.m.abdellah@ieee.org*
[2]Ayman Eldeib is an Associate Professor with the Department of Biomedical Engineering, Faculty of Engineering, Cairo University, Egypt.
[3]Mohamed I. Owis is an Assistant Professor with the Department of Biomedical Engineering, Faculty of Engineering, Cairo University, Egypt.
Authors Emails: [§]*eldeib@ieee.org*, [‡]*mohamed.i.owis@ieee.org*

978-1-4244-9270-1/15/$31.00 ©2015 IEEE

nisms between CUDA and OpenGL is crucial to leverage the performance of the rendering pipeline [9]. Demir *et al.* [10] have presented a hybrid pipeline for computing and displaying electromagnetic fields using CUDA and OpenGL.

In this work, we present an accelerated GPU-based X-ray rendering pipeline that splits the DRR rendering load between a CUDA compute and an OpenGL rendering context using bindless texture objects. The pipeline is applied to generate high resolution DRRs for several medical volumes acquired by different imaging scanners. The performance benchmarks of the pipeline are then demonstrated and optimized.

This article is organized as follows. Section II reviews the theory of X-ray volume rendering. In Section III, we discuss the rendering algorithm and demystify the implementation details. In Section IV, the rendering results are presented and the pipeline performance profiles are discussed. Finally, Section V concludes the paper and highlights some extensions for the presented rendering pipeline.

## II. Theory: X-ray Transform

Given a three-dimensional volume $V(x, y, z)$, the DRR can be easily computed using X-ray transform. This transform evaluates the direct projection of the volume to a two-dimensional plane. Mathematically, the X-ray transform (or John transform) of this volume, denoted by $\mathcal{X}$, is defined on the two-dimensional set of all the one-dimensional line integrals of the volume $V$ [11], [12]. Assuming a line $L$, defined by the parametric formula $L = p + \omega t$, where $p = (p_x, p_y, p_z) \in \mathbb{R}^3$ is a point on the line and $\omega$ is a unit vector along the projection direction, the X-ray transform of the volume $V$ on $L$ is expressed as

$$\mathcal{X}[V(L)] = \int_L V = \int_{-\infty}^{\infty} V(p + \omega t)\mathrm{d}t \qquad (1)$$

As seen in Figure 1, this transform is a direct mapping of each line L in the volume V to a real value that specifies the integration of the data of V along L onto the image plane.

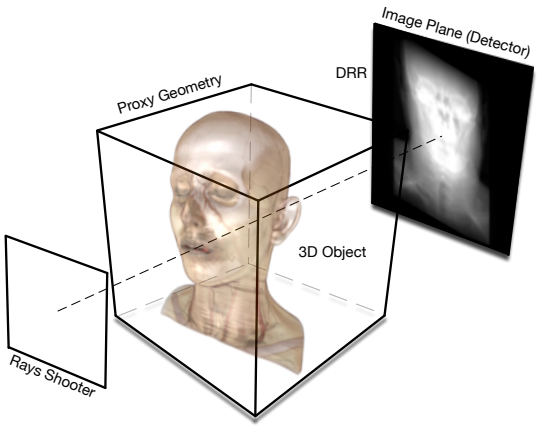



Fig. 1. DRR reconstruction by projecting three-dimensional volume data to the image plane using X-ray transform.

## III. Rendering Algorithm and Pipeline Implementation

### A. DRR Rendering Algorithm

Rendering DRRs is usually implemented either based on a *direct volume rendering* method such as ray casting, ray marching, splatting or a *frequency-domain-based* one such as Fourier volume rendering [13], [14], [15]. The ray marching algorithm is advantageous over ray casting as it does not require computing the intersection points between the rays and the volume analytically. Instead, the ray keeps marching in the space and at each step a binary test is executed to determine if the ray intersects the volume or not. Our rendering pipeline is designed based on the ray marching algorithm. This technique works by shooting virtual rays to propagate within the extent of the volume towards the image plane. For each pixel in the image plane, if the ray intersects the proxy geometry that encloses the volume, it integrates the voxel values during the propagation. The algorithm is summarized in Algorithm 1.

**Algorithm 1** DRR Generation with *Ray Marching*

```
for pixel(x, y) do
    pixel = 0
    Send ray to the scene
    if ray intersects the volume then
        for sample along the ray do
            Evaluate p at sample on ray
            value = volume(p)
            pixel = pixel + value
            if ray exits volume then
                break
```

### B. Pipeline Implementation

The pipeline is designed to allow seamless and intuitive extensibility. A high level overview of the pipeline is illustrated in Figure 2. It supports loading volume data in different file formats (*.dicom*, *.raw* and *.img/.hdr*). During the initialization step, the volume is uploaded to the device memory in a *cudaArray*. The advantage of using *cudaArrays* over linear memory is their opaque layout that is optimized to three-dimensional locality and in turn, the device can operate on three-dimensional blocks instead of one-dimensional rows [16]. Then, the bindless texture object is created and initialized to tri-linear filtration mode and normalized coordinates. The rendering kernel is then executed and writes the projection image to the global memory. After the termination of the rendering kernel, an optional post-processing kernel is activated to invert the image.

The rendering kernel is designed to run independently of any OpenGL calls. However, OpenGL is used in the pipeline for two reasons. The first one is to build the model view matrix and upload it to the device constant memory on a frame-basis. After its generation, the DRR image is sent to an OpenGL texture to be displayed relying on the interoperability mechanisms between CUDA and OpenGL.

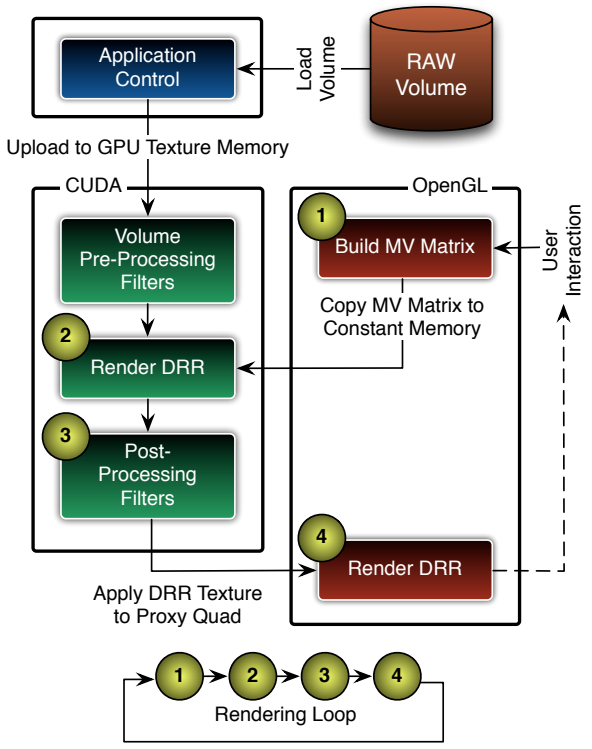



Fig. 2. A simplified block diagram of the DRR rendering pipeline. The CUDA context filters the volume data, computes the DRR image, and post-processes it. The OpenGL context is merely used to build the model-view matrix and renders the final DRR projection. The stages 1, 2, 3, and 4 are executed on a per-frame basis.

The DRR image on the CUDA side is synchronously mapped into a two-dimensional OpenGL texture using Pixel Buffer Objects (PBO). If the shared buffers are mapped to CUDA, they cannot be accessed from the OpenGL context until the completion of the GPU activity to unlock them. Following to this mapping, the image can be directly projected onto a proxy plan for display on an OpenGL widget.

In fact, the advantage of using the texture objects comes with an additional cost. The pipeline requires at least a Kepler-based GPU with compute capability of 3.0 and CUDA 5.0 to run, otherwise the execution of the pipeline will fail. The implementation has also considered rendering volumes with non-unified dimensionality by calculating their scaling factors in every dimension with respect to the largest side and then adjusting the scale of the proxy geometry accordingly. The absence of this step would squeeze the volume into a unit cube and the resulting DRR would be distorted. To reduce the sampling artifacts, the sampling step of the ray is pre-calculated based on the size of the volume. The value of the sampling step is set to be optional where it can be changed on-the-fly. This flexibility allows the user to compensate between the performance and DRR quality if the volume dimensions exceed $512^3$. The pipeline was tested with CUDA 5.5 and 6.0.

## IV. Results & Discussion

Figure 3 shows the resulting reconstructions of three DRRs for different datasets. The datasets are available online and can be freely provided from [17]. The three volumes are not uniform, however, the pipeline is capable of rendering them accurately without distortion.

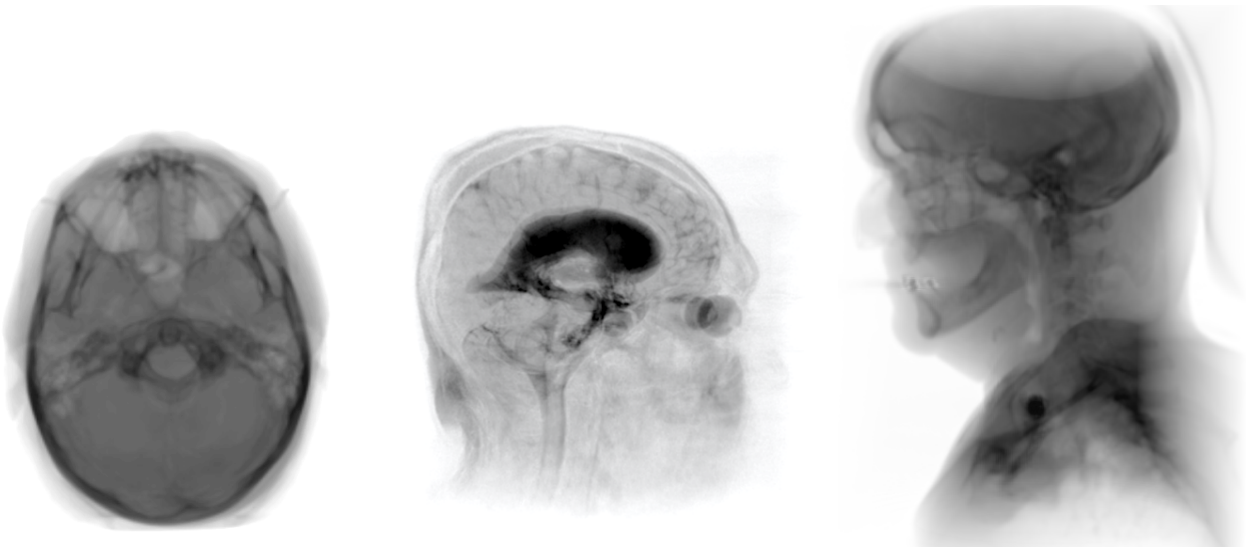
The rendering pipeline was benchmarked on a workstation shipped with an Intel Core i7−4770 chip, 12 GBytes of DDR3 and two GPUs: an NVIDIA GeForce GT 640 (GPU-1) and a GeForce GTX 970 (GPU-2). Figure 4 shows the performance benchmarks for two uniform volumes having $512^3$ and $1024^3$ volume elements on the selected GPUs. The volumes were sampled with uniform step size of $0.01$ within a unit cube defining their proxy geometry and the DRRs were generated with resolutions $512^2$, $1024^2$, $2048^2$ and $4096^2$. To validate the quality of the generated DRRs at this sampling step, the images are compared to similar ones produced with another GPU-based pipeline [18] that uses Fourier transform to compute the DRR. To determine the most optimized CUDA configurations, the dimensions of the CUDA blocks were profiled for $2 \times 2$, $4 \times 4$, $8 \times 8$, $16 \times 16$ and $32 \times 32$ threads per block. All the profiling results were measured with the high precision profiling function *cudaEventRecord* that can resolve the minor differences between the various kernel configurations.

At low frame resolutions, $4 \times 4$ blocks drive extremely high frame rates compared to the other configurations. Increasing the DRR resolution makes larger blocks perform better than smaller ones. At high resolutions, the most optimized block size is found to be $16 \times 16$. This block has comparable performance to the $32 \times 32$ one. However the two volumes have the same spatial extent and reside in the same bounding box, the difference in their sizes clearly affect the frame rate due to their difference in their memory requirements. At DRR resolution of $2048^2$, the profiles express real-time performance and near real-time frame rates for the $512^3$ and $1024^3$ volumes respectively using GPU-1. Using GPU-2, the pipeline line can generate $2048^3$ and $4096^3$ DRRs at 40 and 15 frames per second respectively.

## V. Conclusion & Future Work

In this work, a high performance X-ray volume rendering pipeline is presented to accelerate the generation of high resolution DRRs. This pipeline is optimized to run on the latest NVIDIA GPU architectures exploiting bindless textures, dynamic parallelism and CUDA/OpenGL interoperability. The theory of the technique and the rendering algorithm are briefly reviewed. A high level overview of the pipeline architecture and its implementation details are then explained. The pipeline is employed to create several DRRs for multiple data sets of varying resolutions and dimensions. Finally, the performance of the pipeline is investigated for different CUDA grid and block configurations.

The architecture of the pipeline is designed to be flexibly extended to add support to different post-processing filters relying on the dynamic parallelism of Kepler GPUs. It is



(a) (b) (c)

Fig. 3. Inverted DRR renderings for multiple volume datasets captured with MRI and CT .

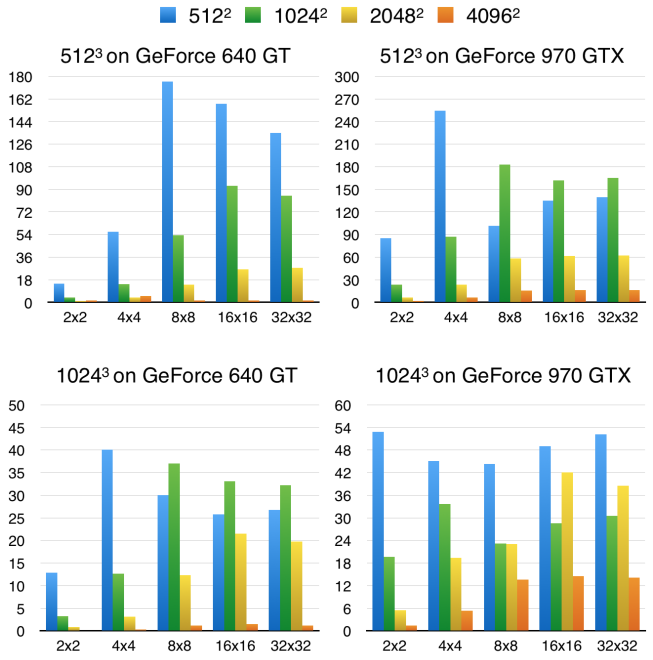



Fig. 4. Pipeline timing benchmarks for two volumes of size $512^3$ and $1024^3$ on two different GPUs. The average profiles are recorded for multiple frame resolutions ($512^2$, $1024^2$, $2048^2$ and $4096^2$) and various grid configuration ($2 \times 2$, $4 \times 4$, $8 \times 8$, $16 \times 16$ and $32 \times 32$). The y-axis represents the frame-rate.

also planned to add multi-GPU support for rendering larger data sets that cannot fit into a single GPU memory.

## References


[1] T. Taxt, A. Lundervold, J. Strand, and S. Holm, "Advances in medical imaging," in *ICPR*, 1998, pp. 505–508.

[2] T. T. Elvins, "A survey of algorithms for volume visualization," *ACM SIGGRAPH Computer Graphics*, vol. 26, pp. 194–201, 1992.

[3] A. Kaufman and K. Mueller, "Overview of volume rendering," *The Visualization Handbook*, pp. 127–174, 2005.

[4] M. A. Westenberg and J. B. T. M. Roerdink, "X-ray volume rendering by hierarchical wavelet splatting." in *ICPR*, 2000, pp. 3163–3166.

[5] D. Russakoff, T. Rohlfing, K. Mori, D. Rueckert, A. Ho, J. Adler, J.R., and J. Maurer, C.R., "Fast generation of digitally reconstructed radiographs using attenuation fields with application to 2D-3D image registration," *Medical Imaging, IEEE Transactions on*, vol. 24, no. 11, pp. 1441–1454, Nov 2005.

[6] Z. Yang, Y. Zhu, and Y. Pu, "Parallel image processing based on cuda," in *Proc. of International Conference on Computer Science and Software Engineering, 2008*, vol. 3, 2008, pp. 198–201.

[7] NVIDIA Corporation, *NVIDIA CUDA C Programming Guide*, July 2013.

[8] J. Foley, *Migrating your code from Tesla Fermi to Tesla K20X, with examples from the QUDA Lattice QCD library*, Microway, Inc., October 2013. [Online]. Available: www.microway.com

[9] J. Stam, "What Every CUDA Programmer Should Know About OpenGL," http://developer.download.nvidia.com/compute/cuda/docs/GTC09Materials.htm, 2009, online, Recorded Session ID 1055.

[10] V. Demir and A. Elsherbeni, "Utilization of CUDA-OpenGL interoperability to display electromagnetic fields calculated by FDTD," in *Computational Electromagnetics International Workshop (CEM), 2011*, Aug 2011, pp. 95–98.

[11] C. A. Berenstein, *X-ray Transform*. Reading, MA: Springer, 2001.

[12] A. Averbuch and Y. Shkolnisky, "3D discrete x-ray transform," *Applied and Computational Harmonic Analysis*, vol. 17, no. 3, pp. 259 – 276, 2004.

[13] C. Johnson and C. Hansen, *Visualization Handbook*. Orlando, FL, USA: Academic Press, Inc., 2004.

[14] M. Abdellah, A. Eldeib, and A. Shaarawi, "Constructing a functional Fourier volume rendering pipeline on heterogeneous platforms," in *Biomedical Engineering Conference (CIBEC), 2012 Cairo International*, Dec 2012, pp. 77–80.

[15] M. Abdellah, A. Eldieb, and A. Sharawi, "Offline large scale Fourier volume rendering on low-end hardware," in *Proceedings of the 7th Cairo International Biolmedical Engineering Conference (CIBEC 2014)*, December 2014.

[16] N. Wilt, *The CUDA Handbook: A Comprehensive Guide to GPU Programming*. Addison-Wesley, 2013.

[17] "The Volume Library," http://lgdv.cs.fau.de/External/vollib/, [Online library for volume visualization datasets, accessed January 2012].

[18] M. Abdellah, A. Eldeib, and A. Sharawi, "High performance GPU-based Fourier volume rendering," *International Journal of Biomedical Imaging*, 2015.