CUDA and OpenCL Implementations of 3D CT Reconstruction for Biomedical Imaging

Saoni Mukherjee†, Nicholas Moore†, James Brock⋆ and Miriam Leeser†

Department of Electrical and Computer Engineering
Northeastern University, Boston, MA 02115
Email: †{saoni, nmoore, mel}@coe.neu.edu, ⋆jbrock@ece.neu.edu

***Abstract*—Biomedical image reconstruction applications with large datasets can benefit from acceleration. Graphic Processing Units(GPUs) are particularly useful in this context as they can produce high fidelity images rapidly. An image algorithm to reconstruct conebeam computed tomography(CT) using two dimensional projections is implemented using GPUs. The implementation takes slices of the target, weighs the projection data and then filters the weighted data to backproject the data and create the final three dimensional construction. This is implemented on two types of hardware: CPU and a heterogeneous system combining CPU and GPU. The CPU codes in C and MATLAB are compared with the heterogeneous versions written in CUDA-C and OpenCL. The relative performance is tested and evaluated on a mathematical phantom as well as on mouse data.**

I. Introduction

Biomedical image reconstruction applications are often not time efficient and thus these can benefit from acceleration. One such biomedical application, CT imaging, has become a very popular medical diagnostic tool. It provides a non-invasive quantification of human body or other living parts for biomedical diagnosis, treatment and research. For diagnostic and treatment interventions, CT imaging methods require high speed reconstructions in real time to remove the chance of interruptions during the treatment of patients. A number of algorithms have been proposed to implement CT reconstruction. Feldkamp, Devis and Kress (FDK) proposed a method that most of today's conebeam CT scanners use. The implementation takes slices of the target, weighs the projection data and then filters the weighted data before backprojecting and creating the final three dimensional construction [1]. The most computationally intensive part is backprojection, which has a complexity of $O(N^4)$ in the spatial domain [4]. This is generally the bottleneck in most software solutions. Many researchers are trying to accelerate conebeam reconstruction using different hardware architectures including Application Specific Integrated Circuits (ASICs) and Field Programmable Gate Arrays (FPGAs) to efficiently use both intensive computing power and speed. However the expensive nature of these boards coupled with less flexibility to reprogram limits their use. Current Graphic Processing Units (GPUs) offer massively parallel processing with the intensive computation necessary for three dimensional conebeam reconstruction.

The main contributions of this paper are:

- Our implementations are compatible with Fessler's image reconstruction tool box [14], a popular toolbox consisting of a collection of open source algorithms for image reconstruction written in MATLAB.
- We compare the performance of two implementations of the same approach, in CUDA-C and OpenCL, to serial and multithreaded C and MATLAB implementations.
- We test our implementations on two types of hardware platforms: CPU and a combination of CPU and GPU. In addition, there are two types of GPUs (NVIDIA and AMD of the same generation) used for evaluating the implementations.

In this paper we provide an overview of the algorithm to reconstruct conebeam tomographic imaging and show how we can move to GPU-based implementations from CPU-based implementations. We also report relative performance using different programming paradigms on different hardware architectures.

II. Related Work

Many researchers have worked on implementing CT reconstruction on GPUs. Mueller et al. [2],[3] list different architectures and programming platforms to explain the use of commodity graphics hardware in accelerating CT reconstruction processes. They propose an OpenGL based implementation that uses the graphics pipeline [2]. It incorporates a load balancing scheme that helps to reduce the computational cost [13]. A similar implementation of conebeam reconstruction using CUDA has been demonstrated by Yang et al. [5]. Another approach is to implement streaming shader-based CT reconstruction that pipelines the process [3]. The work is divided by convolution on the CPU and backprojection on the GPU to reconstruct in faster time. Yang et al. [5], Churchill et al. [6] and Scherl et al. [7] present conebeam CT reconstruction from mobile C-arm units using an NVIDIA device. Scherl et al. implement a CUDA based reconstruction and compare with a Cell-based implementation [7]. Some researchers have introduced other strategies to accelerate conebeam reconstruction. For example, Grass et al. [9] use a rebinning strategy that rearranges and interpolates conebeam projections to turn these into parallel beam projections. A similar method has been used by Li et al. [8] to implement faster backprojection for CT reconstruction. The advantage of

this approach is that it discards redundant information which could affect the noise behavior of the reconstruction.

Our approach can be seen as a generalization of previous work. The implementation is divided into two platforms by convolution on the CPU and backprojection on the GPU [3]. However the load balancing scheme has not been implemented as it does not conform to our goal of reconstructing the whole volume in faster time [13]. Ikeda et al. demonstrate that fragment culling increase the speedup, but it is not possible to get acceleration of the complete reconstruction because the idea is limited to using voxels within a region of interest, culled from rendering. In addition, we used rebinning based on [9] to accelerate the process and improve the quality of the output image. In our implementation, we consider each pixel to be independent and loaded the full volume and all projections consecutively on the GPU. In contrast, Noël et al. [17] consider all projection, but part of the volume. We transfer the whole projection data to the GPU at an early stage and transfer the reconstructed volume back to the CPU at the end. On the GPU a number of threads are launched that can compute on each and every pixel since they can be independently mapped to the final volume.

III. Conebeam computed tomography

Figure 1 shows the schematic drawing of a conventional three dimensional conebeam CT with flat panel detector. Generally a scanner rotates a two dimensional detector or sensor around a patient or object and captures data in it. This process is called conebeam scanning. In this process, the trajectory of the source is circular and each horizontal row of detector values is ramp filtered and considered as a two dimensional object. Then the filtered projections are backprojected along the original rays. During the process of acquiring scanned data, the x-ray source moves in a circular orbital path of radius r and the detector panel moves in the same motion along with the source. The detector plane lies perpendicular to the rotational axis of the x-ray source. It produces a set of projections $P_1, P_2, ..., P_K$ at K discrete positions of the source with uniform angular spacing. Sometimes there are mechanical limitations to completing a full rotation.

Feldkamp et al. [1] published the first algorithm on conebeam reconstruction in 1984. The reconstruction is conceptualized as a weighted backprojection. It is carried out in two stages. In the initial step, the raw data is individually weighted and ramp filtered to produce filtered projections $Q_1, Q_2, ..., Q_K$. The projections are collected at a distance d' with angle θ_n where $1 \leq n \leq K$. d_i is the distance between the volume origin and the source. Let $F(x, y, z)$ denote the value of voxel (x, y, z) in volume F, as shown in figure 2. The xyz space is the volume and uv represents the projections that are to be backprojected to the volume. Then in backprojection, the volume F is reconstructed using the following equations [10]:

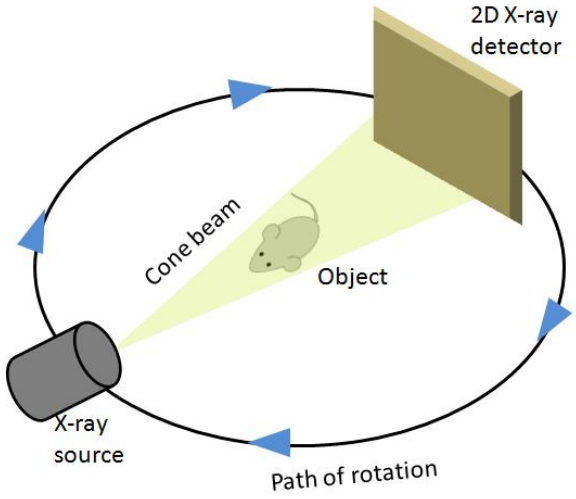

Fig. 1. Scheme of conventional 3D conebeam CT

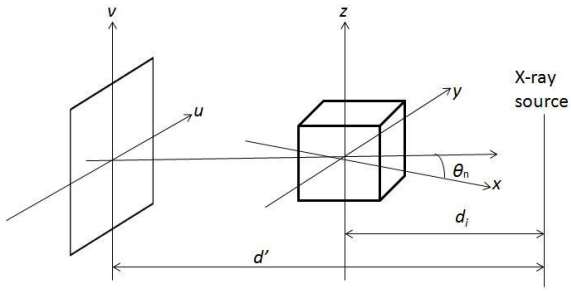

Fig. 2. Co-ordinate system for backprojection

$$F(x,y,z) = \frac{1}{2\pi t}\sum_{i=1}^{t} W_2(x,y,i)Q_i(u(x,y,i), v(x,y,z,i)), \tag{1}$$

where $W_2(x, y, n)$ represents the weight value and $u(x, y, n)$ and $v(x, y, z, n)$ represent the co-ordinates.

$$u(x,y,i) = \frac{d'(-x\sin\theta_i + y\cos\theta_i)}{d_i - x\cos\theta_i - y\sin\theta_i)}, \tag{2}$$

$$v(x,y,z,i) = \frac{d'z}{d_i - x\cos\theta_i - y\sin\theta_i)}, \tag{3}$$

$$W_2(x,y,i) = \frac{d_i}{d_i - x\cos\theta_i - y\sin\theta_i)}. \tag{4}$$

IV. Implementation

A. GPU Architecture and GPGPU

GPUs were originally designed for processing graphics and generating high quality games, but the increase of performance in these units is a perfect fit for scientific computing. The fact that GPUs are increasingly flexible as well as the development of GPGPU (General Purpose GPU) languages contribute to higher peak performance exceeding the CPU in

TABLE I
PERFORMANCE OF DIFFERENT IMPLEMENTATION OF THE METHOD (IN SECONDS)

Dataset	Programming paradigm	Hardware	Time to run Backprojection	Total time
Phantom	MATLAB	Intel Xeon W3580	17.02	17.09
Phantom	C	Intel Xeon W3580	1.36	1.44
Phantom	C with OpenMP	Intel Xeon W3580	0.32	0.33
Phantom	OpenCL	NVIDIA Tesla 2070	0.01	0.11
Phantom	OpenCL	AMD Raedon HD 5870	0.1	0.16
Phantom	CUDA	NVIDIA Tesla 2070	0.01	0.1
Mouse scan	MATLAB	Intel Core i7	8440.1	8443.33
Mouse scan	MATLAB PCT	Intel Core i7	5556.49	5559.9
Mouse scan	C	Intel Xeon W3580	4477.33	4483.83
Mouse scan	C with OpenMP	Intel Xeon W3580	1929.90	1932.27
Mouse scan	OpenCL	NVIDIA Tesla 2070	67.07	91.44
Mouse scan	CUDA	NVIDIA Tesla 2070	42.95	55.47

many problems. GPUs have many parallel cores that can run simultaneously. Each core can run many threads. So at a given point, thousands of lightweight threads are run on the GPU in parallel. Problems with data-parallel computation can be accelerated with the millions of threads available on a GPU. The programming model is SIMD (Single Instruction Multiple Data). Biomedical image processing applications often have a great deal of parallelism and CT reconstruction has inherent features that can be parallelized. Sequential parts can be run on the CPU and computationally intensive parallel parts can be accelerated by running on the GPU. Users across science and engineering are achieving 10x or better speedup by using GPUs.

In this paper, we demonstrate the implementation of back-projection on two types of CPU architectures: CPU and GPU. We have used two CPUs to collect the data- Intel Xeon W3580 processor with 4 cores and Intel Core i7 quad-core processor. The GPUs we have used are NVIDIA Tesla C2070 and AMD Radeon HD 5870 graphics card. For NVIDIA Tesla devices each multiprocessor can have a maximum of 1536 resident threads and there are 14 streaming multiprocessors on the C2070 [12]. So the theoretical limit on the number of threads in flight at once is 21,504. The AMD 5800 series graphics card can run up to 31,744 threads concurrently [16]. Note that these two graphics card are of the same generation.

We have used two GPGPU languages to implement the CT reconstruction, CUDA (Compute Unified Device Architecture) C and OpenCL. Both provide a software environment for writing parallel code that can be run in millions of threads on the GPU. CUDA-C was developed by NVIDIA and runs only on NVIDIA GPUs, whereas OpenCL is developed by the Khronos group and runs on several platforms including AMD, Intel, NVIDIA etc. NVIDIA provides optimized libraries along with CUDA-C, which often results in a better performance. CUDA-C and OpenCL both support heterogeneous computing with separate host code and device code. They have several advantages over other low level GPU programming languages. They require minimal extensions to C/C++ programs. Serial host code runs in one host thread and parallel kernel code runs in several device threads across multiple GPU blocks. The access to arbitrary addresses in device memory is allowed in CUDA and OpenCL. User managed shared memory is shared among threads in a block. The accuracy of data may be as important in biomedical applications as speed. It has been shown that the results provided by GPU may have better precision over serial CPU code for floating point values [15].

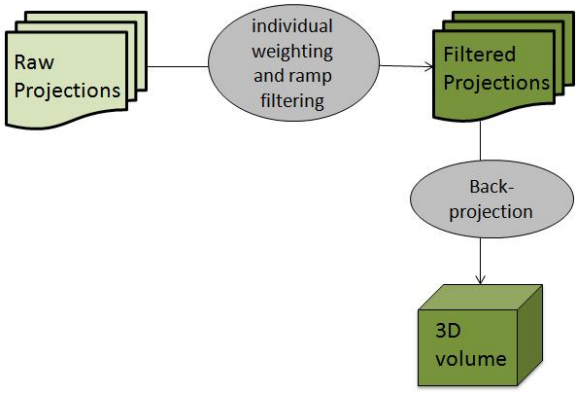

Fig. 3. Overview of serial CPU implementation

B. From CPU to GPU

Figure 3 shows the overview of the FDK method which is divided into three steps: weighting, filtering and backprojection. The first step of the FDK algorithm can be further divided into two parts, weighting and ramp filtering. Weighting includes cosine weighting and short-scan weighting. The weighted projections are then filtered. The last part of the FDK method, backprojection, is the most computationally intensive part.

We start with a serial CPU implementation. Then we parallelize the implementation by using a multi-threaded C program with OpenMP constructs. To move to the next step towards GPU-based implementation from the CPU code, the first thing to be considered is memory transfer. As memory transfer from host to device is expensive, transferring the whole data to GPU before start of computation and transferring back after the final volume is reconstructed is a good idea to save memory cycles. In the filtering stage, different pixels for the same projection can be simultaneously filtered as there is no dependency between any two pixels. Furthermore, different

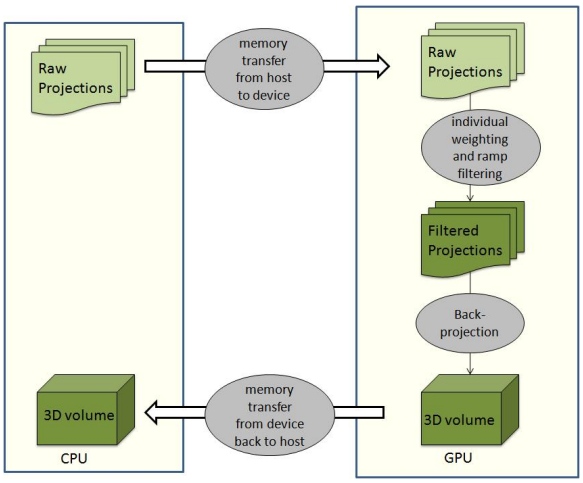

Fig. 4. Overview of GPU implementation

projections can also be simultaneous filtered as they do not have any dependency. Although the next step, backprojection, takes the most time to complete, different voxels are independent and can be processed simultaneously. Figure 4 shows the processing on the GPU including data transfer. In our implementation, we have divided the processing into three parts: weighting, filtering and backprojection. After moving all the projection data to GPU memory, separate kernel calls are launched to execute these three steps. Although the kernel calls are issued in a non-blocking manner, they are executed in series as each step needs to finished before the next can begin.

V. Results

We have implemented the method and evaluated the performance using two datasets. One is a mathematical phantom of input data size 64×60 pixels with 72 projections to get a final volume of $64 \times 60 \times 50$ voxels and the second is a mouse scan of 512×768 pixels with 361 projections. The dimensions of the output volume is $512 \times 512 \times 768$. A single projection of the phantom is shown in Figure 5 and the animal scan is shown in Figure 6. Note that the code depends on the size of the data, not the content. The method is implemented in C, C with OpenMP constructs, CUDA-C and OpenCL and compared with the image toolbox provided by Fessler [14] implemented in MATLAB and a multi-threaded version of the same implementation using MATLAB PCT (Parallel Computing Toolbox) with a poolsize of 8. Performance is reported using an Intel Xeon W3580 quad-core processor with 3.33GHz speed and Intel Core i7 quad-core processor with a 3.4 GHz speed. The graphics that are used are NVIDIA Tesla C2070 and AMD Raedon HD 5870. The number of cores in the CPU are not relevant in the context of the serial application as the code is not parallelized and it runs on a single thread. However the multithreaded C code runs concurrently on 4 threads. Adding more threads introduces overhead and does not improve performance. The performances of all the implementations is listed in Table I.

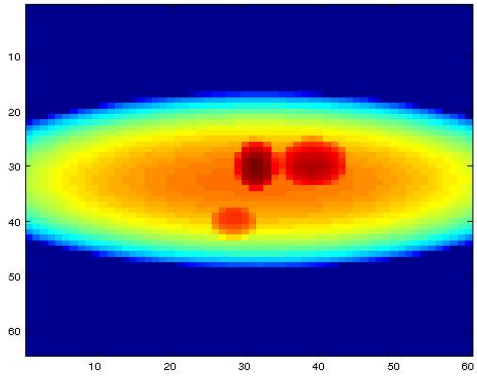

Fig. 5. A single projection of the mathematical phantom

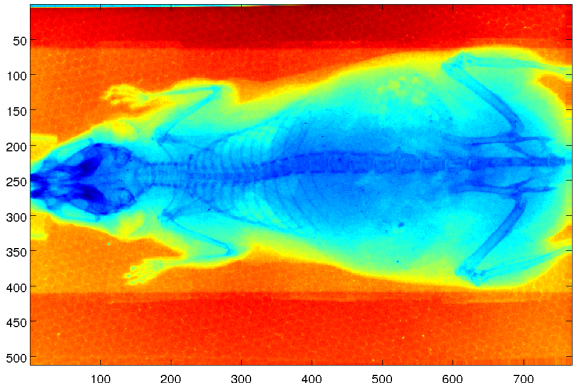

Fig. 6. A single projection of the mouse scan

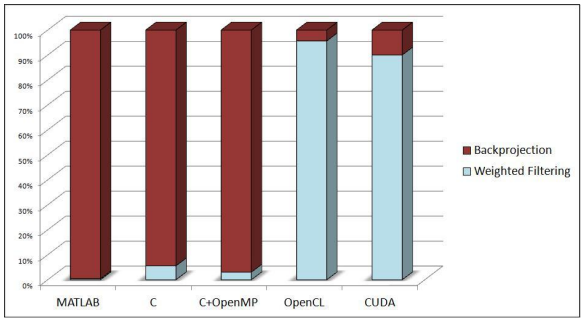

Fig. 7. Relative times taken by two phrases of FDK in different implementations

It is evident that backprojection takes almost 99.9% time of the total time in the serial MATLAB code. The relative times taken by the two phrases, weighted filtering (total of weighting and filtering) and backprojection, are shown in Figure 7. This is the motivation of this work. Backprojection has been parallelized. The multi-threaded MATLAB implementation of backprojection shows a speedup of 1.5x over serial MATLAB in an Intel Core-i7 quad-core processor with 3.40 GHz speed when the implementations are run with the mouse scan data. The multithreaded C implementation of backprojection shows a speedup of around 2x over serial C. OpenCL gives another

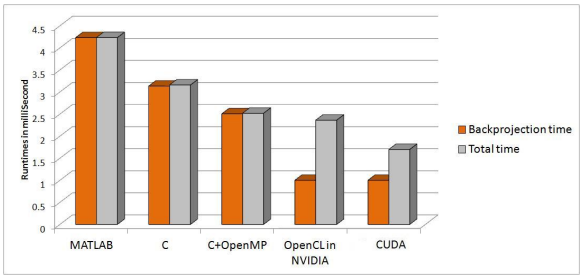

Fig. 8. Runtime for Phantom data of different implementations of FDK method

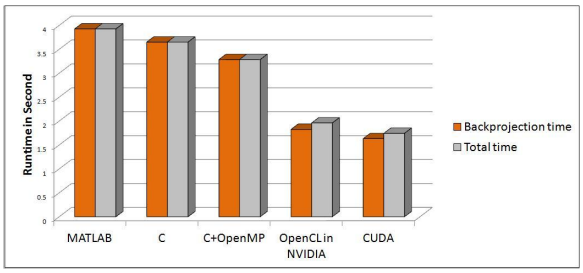

Fig. 9. Runtime for Mouse data of different implementations of FDK method

30x (approximately) and CUDA gives a 50x (approximately) speedup over multi-threaded C for the same dataset. The GPU versions of the code show a speedup of around 80x (OpenCL) and 130x (CUDA) over the multithreaded MATLAB implementations when they are run on the same NVIDIA GPU. CUDA gives a better speedup than OpenCL because the results are taken on a NVIDIA GPU and CUDA is better supported on NVIDIA GPUs than OpenCL [11]. The libraries that are provided with CUDA are optimized to run faster. The FDK method uses a few forward and backward FFTs (Fast Fourier Transform) to implement ramp filters. In OpenCL, the FFTs have been written, but the CUFFT libraries used in CUDA are optimized and provide up to a 10x speedup [12]. Figure 8 and 9 show the runtime taken by different implementations. Runtime is shown on a logarithmic scale. The same OpenCL code is run on NVIDIA and AMD GPUs. A run of the same OpenCL code with the mathematical phantom takes 0.11 seconds to reconstruct the image on an NVIDIA GPU, whereas it takes 0.16 seconds on the AMD GPU of the same generation. We suspect that the speedup is comparatively less in the AMD GPU because the code is not optimized enough to run on AMD GPU architecture. The runtime of different kernels of the same OpenCL code when it is run on NVIDIA and AMD GPUs with the mathematical phantom is shown in Table II. Here the mouse data could not be tested because the AMD GPU did not have enough memory to allocate all the projections and final volume. Note that every kernel runs faster on the NVIDIA architecture.

VI. Future Work

Although the current execution settings produce significant speed-ups on GPUs, the runtime can be still optimized. After backprojection is parallelized, the new bottleneck is the weighted filtering step. This needs to be sped up more. In addition, only a subset of the number of launch kernel configurations have been tested so far. The number of threads are arbitrarily chosen from a small set of tests. These issues will be investigated with auto-tuning. The data sizes that are seen so far can be accommodated in the GPU memory, but for larger data sizes, streaming can be added to the current implementation. However that may result in significant overhead. Overlapping communication and computation will be investigated.

TABLE II
Performance of the same OpenCL code on NVIDIA and AMD GPUs (in milliseconds)

GPU	Kernel	Time
NVIDIA	Weighting	2.25
	Filtering	89.62
	Backprojection	14.07
AMD	Weighting	14.70
	Filtering	123.23
	Backprojection	19.68

VII. Conclusion

We have presented a faster way to reconstruct conebeam projections in a GPU-enabled system based on the FDK method. Our results show that, on an NVIDIA GPU, CUDA-C outperforms the OpenCL implementation due to better support for CUDA. The experimental results show that the CUDA code takes 42.95 seconds to backproject a $512 \times 512 \times 768$ voxel volume from 361 projections of 512×768 pixels, which is approximately 200x faster than the single-threaded implementation in MATLAB, approximately 100x faster than the single-threaded implementation in C and around 50x faster than the multi-threaded implementation C with OpenMP constructs. The scanner takes 8.42 seconds to collect the mouse scan data and generating the complete volume takes around 43 seconds using the CUDA-C implementation.

VIII. Acknowledgments

This work was supported in part by the National Science Foundation Engineering Research Centers Innovations Program, Biomedical Imaging Acceleration Testbench (Award Number EEC-0946463). This work is funded in part by a gift from Mathworks

We thank Drs. Ralph Weissleder and Sarit Sekhar Agasthi, Massachusetts General Hospital for providing the mouse scan data.

References

[1] L. A. Feldkamp, L. C. Davis, J. W. Kress, Practical cone-beam algorithm, *J. Opt. Soc. Am.*,Volume 1(A), (1984).
[2] F. Xu, K. Mueller, Real-time 3D computed tomographic reconstruction using commodity graphics hardware, *Physics in Medicine and Biology*, 52(12) (2007).
[3] K. Mueller, F. Xu, N. Neophytou, Why do commodity graphics hardware boards (GPUs) work so well for acceleration of computed tomography?, *Proceedings of the SPIE*, Volume 6510, (2007).
[4] K. Mueller, F. Xu, Practical consideration for GPU-accelerated CT, *IEEE Int. Symp. Biomed. Imaging*, (2006).

[5] H. Yang, M. Li, K. Koizumi, H. Kudo, Accelerating Backprojections via CUDA Architecture, *9th International Meeting on Fully Three-Dimensional Image Reconstruction in Radiology and Nuclear Medicine*, Volume 9, (2007).
[6] M. Churchill, G. Pope, J. Penman, D. Riabkov, X. Xue, A. Cheryauka, Hardware-accelerated cone-beam reconstruction on a mobile C-arm, *Proceedings of the SPIE*, Volume 6510, (2007).
[7] H. Scherl, B. Keck, M. Kowarschik, J. Hornegger, Fast gpu-based ct reconstruction using the common unified device architecture (cuda), *Nuclear Science Symposium Conference Record*, (2007).
[8] M. Li, H. Yang, K. Koizumi, H. Kudo, Fast cone-beam CT reconstruction using CUDA architecture, *Medical Imaging Technology* 25(4), (2007).
[9] M. Grass, T. Kohler, R. Proksa, 3D cone-beam CT reconstruction for circular trajectories, *Physics in Medicine and Biology* 45(2), (2000), 329347.
[10] F. Ino, S. Yoshida, K. Hagihara, RGBA Packing for Fast Cone Beam Reconstruction on the GPU, *Proc. of SPIE*, Volume 7258, (2009).
[11] P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, J. Dongarra, From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming, September 2010.
[12] NVIDIA corporation, NVIDIA CUDA C Programming Guide, CUDA Toolkit 4.1.
[13] T. Ikeda, F. Ino, K. Hagihara, A code motion technique for accelerating general-purpose computation on GPU, *Proc. 20th IEEE Int'l Parallel and Distributed Processing Symp.*, (2006).
[14] Fessler's image reconstruction toolbox, http://www.eecs.umich.edu/∼fessler/irt/fessler.tgz.
[15] D. Yablonski, Numerical Accuracy Differences in CPU and GPGPU Codes, Masters thesis, Northeastern University.
[16] https://en.wikipedia.org/wiki/AMD_FireStream
[17] PB Noël, A Walczak, KR Hoffmann, J Xu, JJ Corso, S Schafer, Clinical Evaluation of GPU-Based Cone Beam Computed Tomography, *Proc. of High-Performance Medical Image Computing and Computer-Aided Intervention (HP-MICCAI)*, (2008).