Why do commodity graphics hardware boards (GPUs) work so well for acceleration of computed tomography?

Klaus Mueller, Fang Xu, Neophytos Neophytou

Klaus Mueller, Fang Xu, Neophytos Neophytou, "Why do commodity graphics hardware boards (GPUs) work so well for acceleration of computed tomography?," Proc. SPIE 6498, Computational Imaging V, 64980N (28 February 2007); doi: 10.1117/12.716797

SPIE.

Event: Electronic Imaging 2007, 2007, San Jose, CA, United States

Why do Commodity Graphics Hardware Boards (GPUs) work so well for acceleration of Computed Tomography?

Klaus Mueller, Fang Xu, and Neophytos Neophytou

Computer Science, Stony Brook University, Stony Brook, NY 11794, USA

ABSTRACT

Commodity graphics hardware boards (GPUs) have achieved remarkable speedups in various sub-areas of Computed Tomography (CT). This paper takes a close look at the GPU architecture and its programming model and describes a successful acceleration of Feldkamp's cone-beam CT reconstruction algorithm. Further, we will also have a comparative look at the new emerging Cell architecture in this regard, which similar to GPUs has also seen its first deployment in gaming and entertainment. To complete the discussion on high-performance PC-based computing platforms, we will also compare GPUs with FPGA (Field Programmable Gate Array) based medical imaging solutions.

Keywords: Computed Tomography, high-performance computing, graphics hardware, GPU, cone-beam reconstruction

1. INTRODUCTION

The thirst for visual and physical realism in computer games, backed by heavy consumer spending and enabled by a growing sophistication in VLSI design and manufacturing, has given rise to an incredible performance growth in PC-scale computing platforms. Peak performances of 500 GFlops (10^9 floating point operations/s) and more are now possible, which is 100 times greater than the peak performance of the 16-processor Cray C90 supercomputer that ruled the national labs just a decade ago, in form of a closet-size computer with a $500k price tag. In contrast, the most popular of these PC-scale computing platforms, the boards hosting a Graphics Processing Unit (GPUs), fit into the PCI slot of any standard desktop computer (or even a laptops) and are available for $500 or less at any neighborhood computer outlet.

In fact, due to the to the increasing levels of programmability and flexibility, these computing platforms have also found use in a much broader range of general computing applications, including those formerly only achievable with supercomputers. GPU-clusters targeted for large-scale scientific computing have been emerging [1][2], and the folding@home [3] distributed supercomputing effort will also soon utilize the legions of idle GPUs to gain better understandings of the protein folding process. The use of GPUs for general-purpose computing has become popularly known as General Purpose GPU (GPGPU), and an extensive website www.gpgpu.org logs many of these works. For example, GPGPU has enabled the acceleration of computations in domains as diverse as database processing [ref], computer vision [4], computational geometry [5], sorting [6], and also medical imaging [7][8], which is the subject of this paper.

GPUs are, in some ways, similar to the vector processing units of the Cray supercomputers of the past which, in contrast to the sequential instruction-driven Von Neumann style data processing of CPUs, only required the decoding of one single instruction for a long vector of data. But unlike this traditional vector processing hardware, GPUs can perform multiple operations per data item which is more efficient in terms of memory bandwidth. In this regard, GPUs share more common features with the later *streaming architectures*. Nevertheless, despite these close relations to vector and stream processors, the true ancestors of GPUs are the geometry and rasterization engines of the Silicon Graphics (SGI) workstations of the 1990s. Recognizing the massive amount of identical operations that exist within graphics applications, consisting of vertex transformations and shading, followed by scan-conversion with texture mapping interpolation, an SGI Infinite-Reality workstation featured four heavily pipelined geometry and raster engines each. Both employed the SIMD (Same Instruction Multiple Data) data processing model, which is also used by GPUs today. However, a feature that was completely lacking in these early SGI platforms was the ability to freely program the SIMD processors. Rather, all one could do was set the parameters used in the graphics processing operations, such as lighting

Computational Imaging V, edited by Charles A. Bouman, Eric L. Miller, Ilya Pollak
Proc. of SPIE-IS&T Electronic Imaging, SPIE Vol. 6498, 64980N, © 2007 SPIE-IS&T · 0277-786X/07/$18

model, z (depth)-buffer mode, and so on. To enable this communication, SGI founded the OpenGL graphics language which is still in use today, now accompanied by its rival DirectX.

The lack of programmability impeded the use of SGI hardware for serious GPGPU applications. Nevertheless, it was medical imaging that was chosen to be one of the few non-graphics applications to be run on that platform. In 1992, Cabral, Cam, and Foran [9] described how one could accomplish both volume rendering and inverse volume rendering, that is, CT reconstruction (via filtered backprojection) on SGI texture mapping hardware. Later, Mueller and Yagel [10] employed the same platform for iterative reconstruction (with SART), which used (X-ray) volume rendering for forward projection and inverse volume rendering for backprojection. Since both operations can be implemented fairly easy with OpenGL instructions, using projective textures [11] for backprojection from arbitrary angles, the main limitation was imposed by the lack of floating point precision. This required the backprojection accumulations to be done on the CPU, which incurred major data communication delays. Fortunately, these could partially be hidden by splitting longer data words into the 4 (RGBA) color channels. Finally, the amount of texture memory available to store the data and the reconstruction result was also very limited, on the order of MBs.

Today's GPUs have none of these shortcomings. They are generally programmable, have full IEEE single-precision floating point arithmetic, and due to the PCI-Express bus also have data fast transfer rates from main memory to GPU, where the data transfer may overlap with ongoing computations. They also feature large texture memories of 512 MB-1 GB, which can hold reconstruction volumes of size 512^3-1024^3 at floating point precision. Finally, they can be run in dual and quad board configurations on a single PC or a shuttle PC. And what is most relevant for this paper, they allow a 512^3 volume to be reconstructed at full floating point precision in real time at a bandwidth of 30 projections/s.

Our paper is structured as follows. Section 2 will provide some general comments on GPU and stream-based computing and will also discuss other high-performance architectures. Section 3 will elaborate further on prior and related work using PC-grade high-performance architectures for CT reconstruction. Following, Section 4 will briefly discuss the theory of filtered backprojection via Feldkamp's algorithm. Then, Section 5 will discuss in further detail both architecture and programming model of GPUs, and Section 6 will describe how this can be used for high-performance CT reconstruction. Section 7 will present results obtained with our GPU-based system, called RapidCT. Finally, Section 8 will discuss the results and put them in perspective with other work, ending with conclusions.

2. SOME GENERAL COMMENTS ON GPU AND STREAM-BASED COMPUTING

The overall target of GPUs has not changed since the early SGI architectures, that is, to provide dazzling graphics effects for entertainment, visual simulations, and engineering. This task implies feeding a rectangular array of screen pixels with a massive amount of data, consisting of textures and geometry. Such a process can be elegantly modeled as a continuous data *stream*, which is consumed by a pipeline of processing *kernels* at a minimum of data dependencies. Since all pixels are being composed via the same underlying forward-streaming computation, an inherent data-parallel processing model can be employed – SIMD. It is this SIMD model that makes processor design simple and raises the percent chip area dedicated for arithmetic. It also bolsters the number of transistors and encourages rapid growth. Take for example the most recent G80 Nvidia GPU core, now available as the GeForce 8800 GTX. It has 681M transistors based on a 0.09 micron process. Just half a year earlier the G71 core (sold as the GeForce 7900 GTX) was released, which had 278M transistors using the same 0.09 micron process. The benchmark performance of the 8800 GTX is about double of that of the 7900 GTX. In relation to the projected growth of general purpose CPUs, which is predicted by Moore's law and amounts to an 18-month period for a doubling of benchmark performance, this represents a performance growth at a rate of 3 times Moore's law. Further, in contrast to CPUs where much of the chip real estate goes into caches and cache management (for example, the Xeon chip uses 60% of the chip area for this), in GPUs the increasing number of transistors goes right into the arithmetic pipelines. In fact, GPUs have gained much of their speed increases by providing more SIMD pipelines (as well as providing more efficient mechanisms to keep these pipelines busy). For example, while the G71 had 24 pixel pipelines and 8 vertex pipelines (with a 650-750MHz clock and a 256-bit wide bus), the new G80 has a whopping 128 arithmetic pipelines, now unified for better pixel/vertex load balancing [12]. Further, not only is the chip area dedicated to computation higher in GPUs, the total number of transistors is also typically greater. For example, the latest AMD processor, the 64 FX dual core chip, has 227M transistors (on the same 0.09 micron process) which is 1/3 of the G80's transistor count. Again, this can be attributed to the less complex SIMD design of GPUs.

However, GPUs are not a general-purpose processor. They can only live up to their full potential when presented with a computational task that bears the features of the underlying streaming SIMD programming model. A few general rules are presented here:

1. It is better to focus each pipeline onto a single output data element (the pixel or *fragment*) than onto many output data elements (*fragments*). The former is called *gathering*, where many input data are processed to compute a single fragment. The latter is called *scattering*, where a process must spread its computation results onto many fragments. Fortunately, many times one can just reformat the computation to form a GPU-suitable program flow.
2. The same goes for programs that have a large number of conditional statements. For example, Quicksort is a very efficient algorithm (often the most efficient) for sequential CPU platforms, but it is very inefficient for GPUs. Here, it is better to use Bitonic Sort, which is a highly parallelizable form of sorting and runs very fast on GPUs [13].
3. It is also advised to have a sufficient amount of data items available to fill the pipelines and get a data stream going. The memory is not optimized for latency, but for bandwidth, which favors the streaming model. Although the hardware management facilities for fragments waiting for data items have become better (the pipelines are then fed with other waiting fragments having all data available), explicit program-controlled load-balancing of the different pipeline components is still required in many cases.
4. It is also important that GPU programs exhibit a good amount of arithmetic intensity [ref]. Sometimes it is better to compute an intermediate result than to query a lookup table. Just like in any computational platform, memory is still a bottleneck, but GPUs excel (over CPUs) at computational speed.
5. Finally, due to the GPU strong heritage in graphics, where linear interpolation and the compositing/blending of two weighted values are frequent operations, these types of operations are often available in form of fast hardwired circuitry. Thus, it pays to make use of these facilities as much as possible.

Therefore, if a computational task exposes, or can be reformulated to expose, these types of features, then it is likely that it will exhibit the typical speedups achieved with such a port. In order to assess the growth of GPU performance one may compare the increase in the rate of floating point processing, measured in GFLOPS. This is shown in the graph of Fig. 1. We can see that the gap between CPU and GPU performance is widening quickly.

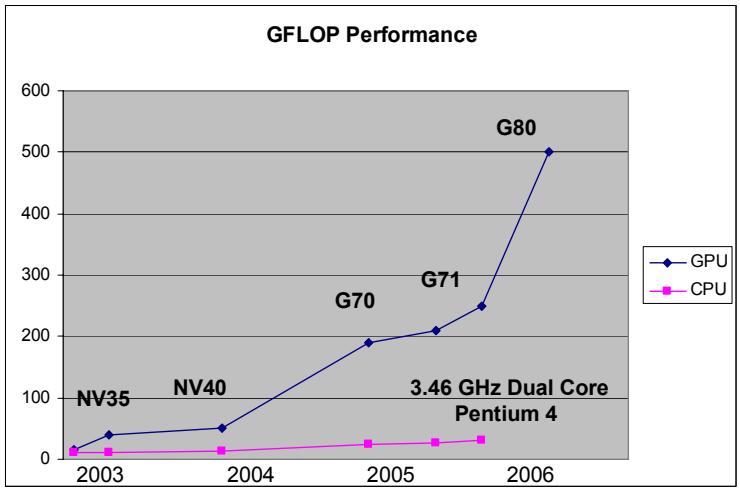

Figure 1: Growth curves of GFLOP performance of GPUs and CPUs.

For this reason, CPU manufacturers, such as AMD and IBM have sought to strike back. AMD recently acquired GPU manufacturer ATI and just released a stream processor card, the R580 CPU with 1GB of memory, which can be programmed using their new CMT (Close-to-Metal) programming interface [14]. Since this card is very similar to the former ATI GPU board, similar performance can be expected. IBM, in collaboration with Sony and Toshiba, has recently unveiled the Cell processor, which consists of one main processor element (PPE) and 8 tightly coupled synergistic 2-way processing elements (SPEs) [15][16]. The SPEs are similar to the processors found on GPUs, but they can be chained together under program control to form a MIMD (Multiple Instruction Multiple Data) configuration. The Cell processor has so far been used in the new Sony Playstation 3, and Mercury, Inc. [17] has also announced products that use the Cell Broadband Engine (BE) in various accelerator configurations. The Cell processor has a quoted clock speed of 3.2 GHz, while the G80 processors run at about half that speed. The peak performance of the Cell processor is 256 GFLOPS, which is far less than that of the G80, possibly due to the fewer number of pipelines. However, while it is tempting to compare GPUs and the Cell just by the number of pipelines, clock speeds, and GFLOP performance, only the particular application and their implementation can really tell which is more favorable to use.

On the other hand, GPU manufacturer Nvidia has also sought to make its processors more available to the general user community. To achieve this goal, they recently introduced CUDA (Compute Unified Device Architecture), an interface that allows users to write high-performance programs for any compute-intensive task in the standard C-language. Previously, users needed to use CG [18], in conjunction with OpenGL or DirectX, which yielded assembler-like code that could be further optimized by hand. In this context, one should also mention Brook [19], a streaming

programming language from Stanford University which extended the programming language C to streams and produced optimized CG code when compiled.

Finally, another way to obtain a high-performance architecture is by configuring a field programmable gate array (FPGA). An FPGA is a semiconductor device consisting of programmable logic components and programmable interconnects. Any basic logic such as AND, OR, XOR, NOT or more complex combinational functions such as decoders or simple math functions can be programmed. FPGAs are generally slower application-specific integrated circuit (ASICs) and can only designs of moderate complexity. Nevertheless, they allow programmers to hardcode a specific algorithm without incurring the overheads of instruction-style programming.

3. PRIOR AND RELATED WORK IN PC-SCALE HIGH-PERFORMANCE CT

For this discussion we have purposely restricted ourselves to approaches that use a single PC or workstation for reconstruction – there are a number of applications that have used PC clusters. The introduction has already mentioned the works by Cabral et al. [9] and Mueller and Yagel [10] to exploit high-end SGI workstations for accelerated CT. We mentioned that the low 12-bit precision of the hardware was particularly limiting in the iterative scenario, and to cope, Mueller and Yagel devised a dual-channel scheme, using the RB (Red Blue) color channels, which enabled a pseudo 16-bit precision and which also enabled some of the accumulations to be done directly in the hardware. The first paper discussing the use of PC-native GPU boards for CT was the one by Chidlow and Möller [20] who implemented emission tomography with OS-EM [21] on an Nvidia GeForce 4 GPU card. Although good speed-ups and quality reconstructions could be achieved, the potential of these solutions remained to be limited, due to the still limited precision and accumulation capabilities. This required many operations still to be performed on the slower CPU, which incurred expensive data transfers. To cope, similar to Mueller and Yagel, they also employed a virtual frame buffer extension by combining the color channels. Soon after, GPUs with full programmability and 32-bit floating point precision enabled a complete GPU-resident CT reconstruction, with both analytical and iterative methods [7] and with large data [8] at a fidelity comparable to CPU-based methods. Following these more fundamental works were a number of papers targeting specific CT applications, all with impressive speedup factors. Kole and Beekman [26] accelerated the ordered subset convex reconstruction algorithm, Xue et al. [23] accelerated fluoro-based CT for mobile C-arm units, and Schwietz et al. [24] accelerated the backprojection and FFT operations employed for MR k-space transforms. Finally, Xu and Mueller [25] used their GPU-accelerated reconstruction framework to enable interactive volume visualization directly from a full set of projection data.

FPGA-based solutions include the cone-beam CT application by Goddard and Trepanier [26], the 9-bit precision parallel-beam application by Leeser et al. [27] and the 16-bit precision cone-beam reconstructor by Li et al. [28]. The performance times are somewhat similar, when normalized to a certain problem size. Goddard and Trepanier reconstruct a 512^3 volume from 300 cone-beam projections in 38.7s using Feldkamp's algorithm, while Li et al. solve the same problem in 33.5s. Extrapolating the parallel-beam results of Leeser et al. to this problem also yields a reconstruction time of 37s.

Recently, the Cell processor was also used for cone-beam reconstruction with Feldkamp's algorithm [9]. Using the Cell-board available from Mercury, Inc., a 512^3 volume could be reconstructed from 512 projections in 13.6 s (8s for our normalized problem size) [30]. In fact, the Mercury board, called a blade, hosts two Cell processors and thus a reconstruction could be achieved in half the time, that is, 6.8s (4s).

4. THE FELDKAMP CONE-BEAM RECONSTRUTCTION AGORITHM

Our framework uses the standard Filtered Backprojection algorithm devised by Feldkamp, Davis, and Kress [9]. Basically, the Feldkamp (FDK) algorithm consists of three stages: projection-space filtering, back-projection, and volume-space weighting. This is captured in the following three equations (see also Fig. 2):

$$f(\boldsymbol{r}) = \frac{1}{4\pi^2} \int_0^{2\pi} \frac{d^2}{(d + \boldsymbol{r} \cdot x_\phi)^2} \hat{P}_\phi(\boldsymbol{r}) d\phi \qquad (1)$$

where

$$\hat{P}_\phi(\boldsymbol{r}) = \hat{P}_\phi(Y(\boldsymbol{r}), Z(\boldsymbol{r})), \quad Y(\boldsymbol{r}) = \frac{\boldsymbol{r} \cdot y_\phi}{d + \boldsymbol{r} \cdot x_\phi} D, \quad Z(\boldsymbol{r}) = \frac{\boldsymbol{r} \cdot z_\phi}{d + \boldsymbol{r} \cdot x_\phi} D \tag{2}$$

and

$$\hat{P}_\phi(Y, Z) = \frac{D}{\sqrt{D^2 + Y^2 + Z^2}} P_\phi(Y, Z) ** g(Y) \tag{3}$$

In these equations, a voxel is denoted by a vector $\boldsymbol{r} = (x, y, z)$ in the (reconstruction) volume space defined by (X_v, Y_v, Z_v), with Y_v being the rotation axis. The elements on the projection (detector) plane oriented at ϕ are represented by $P_\phi(Y, Z)$, where Y and Z represent an element's spatial location in detector coordinates. The vector X_ϕ is orthogonal to the detector plane and connects the detector center with the source S. The two orthogonal vectors Y_ϕ and Z_ϕ complete the 3D coordinate system of the detector space, which is related to (X_v, Y_v, Z_v) by ways of a transformation matrix composed of gantry rotation and possible gantry warp. Finally, the distances from the source to the rotation center O and the detector center are defined as d and D, respectively.

Equation (3) constitutes the projection-space filtering, while equation (2) represents the backprojection operation, that is, the mapping of a voxel to a location on the projection plane and the subsequent assignment of the value interpolated there. Finally, equation (1) integrates the backprojection contributions for a given voxel over all projections, properly weighted in volume space. These three equations constitute a natural decomposition of the FDK algorithm, forming a pipeline consisting of filtering, backprojection, and weighting.

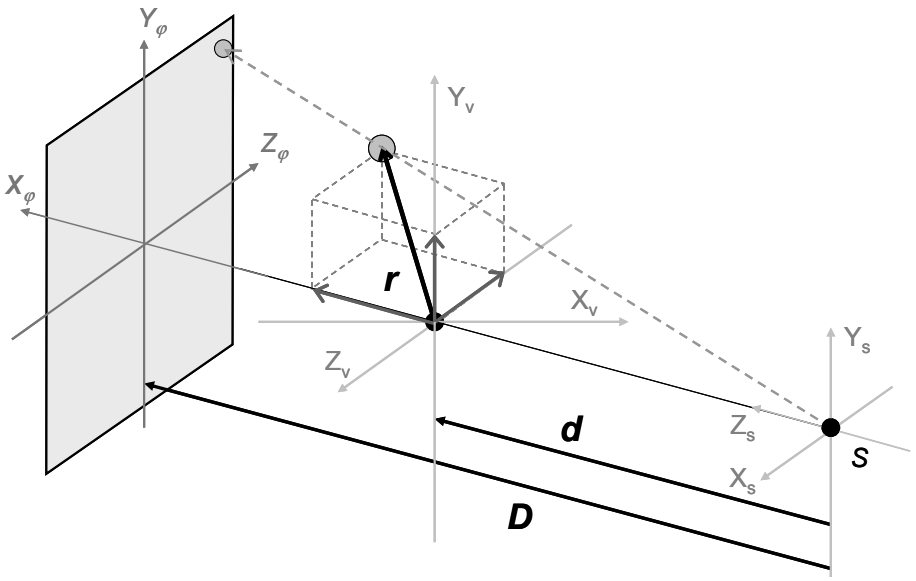

Figure 2: The Feldkamp filtered back-projection algorithm.

5. GPU ARCHITECTURE AND POGRAMMING MODEL

While not overly flexible in terms of program flow, GPUs are extremely powerful when it comes to the repetitive data processing often exhibited in loops. In fact, this form of computational model is very typical for graphical objects, which consist of large meshes of polygons, with each such polygon being defined as a set of three or more vertices. Here, each such vertex is represented as a 3D floating point (x,y,z) coordinate triple. To view such a graphical object from an arbitrary position, all vertices must first be transformed into viewing-space (defined by the viewing direction), orthographically or perspectively, and then reconnected to form the (projected) polygons. If only these connections (or

edges) are displayed, then the object appears as a *wireframe model*. However, if all screen pixels subtended by a given polygon are assigned a value, then a process called rasterization is invoked. More generally speaking, rasterization produces data (called *fragments*), which are associated with the corresponding screen pixels. Also, apart from shading effects, one can add more surface detail by assigning each vertex a certain location in one or more images or *textures*. Each polygon then subtends a closed (polygonal) region in this texture, and the rasterization process provides the coordinates (as part of the fragment) into these textures (Fig. 3a). Since these do not necessarily coincide with a discrete texture pixel, the texture must be interpolated (either with linear or nearest neighbor interpolation) to yield the value assigned to or combined with the screen pixel (Fig. 3b).

Thus there is a strict computational pipeline governing the display of a graphical object: (1) the vertex transformations, (2) the generation of fragments, and (3) the computations imposed on the fragments, such as texture interpolations or more complex tasks, which result in the RGBA colors assigned to the screen pixels. Since screen pixels are numerous and typically independent, a high degree of SIMD parallelism exists. Further, since this basic graphics pipeline has no loop dependencies, the objects simply *stream* across the pipeline, starting as a list of vertices, which generate fragments, which in turn form the basis for computing the visual attributes assigned to the corresponding screen pixels. GPUs are hence a *streaming architecture*, processing massive sets of graphics primitives, i.e., polygons and textures, in a highly parallelized fashion. This dedicated computational model greatly simplifies the control circuitry which intensifies data processing and also enables optimized memory access for maximum throughput with significantly reduced latency.

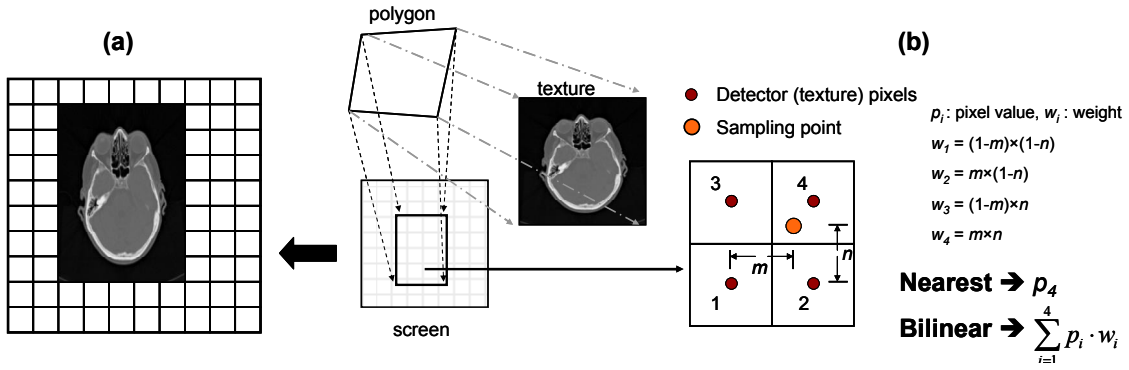

Figure 3: GPU operations: (a) rasterization; (b) interpolation.

More concretely, vertices and fragments that share similar operations constitute the elements of a *stream*, where these similar operations on them are implemented via user-defined *shader programs*, one at the *vertex stage* and one at the *fragment stage* of the pipeline (see the red boxes shown in Fig. 4a). When rendering begins, a stream of vertices generated on the CPU, or stored in GPU *vertex memory*, is passed into the GPU's *geometry processing* stage. These vertices usually carry multiple properties to describe the target object, such as coordinate (xyz), color (RGBA), normal, etc. A *vertex shader* can then be applied to transform and map the 3D coordinates of each vertex to homogenous space and eventually screen space, while possibly complex vertex lighting effects can also be calculated. Next, *rasterization*, which is performed in fast special GPU hardware circuitry, re-assembles each polygon in screen space and quickly fills the space enclosed by it. This generates a stream of fragments, which map to corresponding screen pixels. The rasterizer also linearly interpolates any scalar or vector attribute that may have been associated with the vertices, such as 3D coordinates, texture indices, and others. This fragment stream is processed in the *fragment shader*, which performs a series of user-defined per-pixel calculations. In this effort, the fragment shader also gives rise to another data stream, consisting of textures, which are stored in GPU *texture memory* and provide the data used in the fragment shader. After fragment program completion, the result is written to the corresponding pixels on the screen (or the *framebuffer*).

Textures are the dominating data representation and storage primitive used by GPUs. A texture is essentially a 1-3D array of data. Each texture element can be represented in a certain *format,* ranging from a scalar to a vector of up to four components. In a graphics context, the latter usually describes a Red/Green/Blue/Alpha (RGBA) value, but note that for general-purpose GPU-computing this 4-channel representation can provide an additional 4-way parallelism. This

parallelism can be exploited by packing data sharing common operations together, and good speedups can be obtained since GPUs tend to be very efficient at processing this type of vectors. Current GPUs also support various texture *precisions* ranging from basic 8-bit integer to advanced 32-bit floating point. There are, however, still constraints on the use of textures in certain combinations of formats and precisions. For example, 4-channel 16-bit fixed point textures are not yet supported. Further, for most GPU-based applications, 2D textures are the preferred primitive since they are naturally optimized by the hardware and support efficient read/write actions. 3D textures, in contrast, are used mainly in (read-only) visualization applications, since they do not support direct-write operations. When write-access is desired they are better represented as stacks of 2D textures.

The Nvidia GeForce 7800 GTX provides up to 8 vertex pipelines and 24 pixel (fragment) pipelines and the latest GeForce board, the 8800 GTX, has 128 unified pixel/vertex pipelines. Thus, a computational process also taking advantage of the 4-channel RGBA parallelism can achieve a 96-fold (512-fold) SIMD concurrence. Finally, dual-GPU boards or even quad-GPU designs have also become recently available on the market. These designs use the Scalable Link Interface (SLI) to connect the multiple GPUs together, and such a setup can double or quadruple, respectively, the amount of performance of a single GPU, on a single PC!

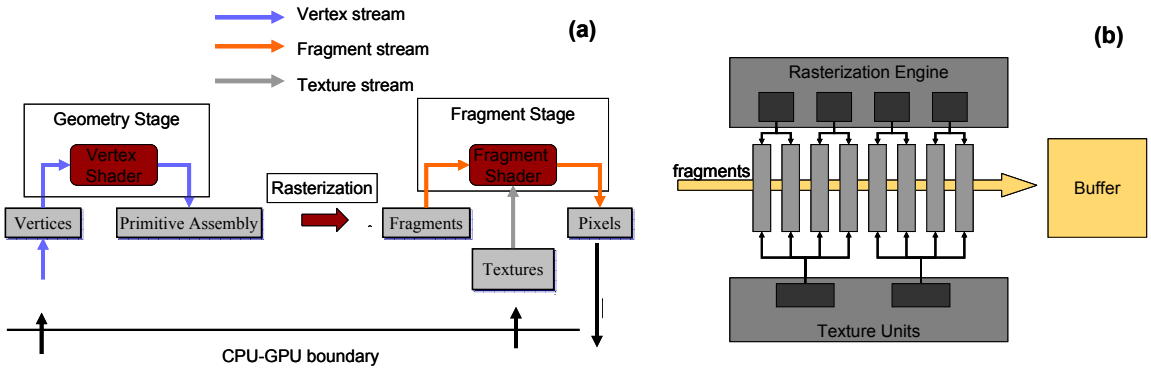

Figure 4: GPU components: (a) graphics pipeline, (b) rasterization and fragment processing stage.

6. GPU-ACCELERATED CT RECONSTRUCTION

In the past section we have described the basic elements of a GPU pipeline. In this section we will describe how we map the three FDK constituents into this framework, which yields a streaming CT application.

6.1 Projection-Space Filtering

We choose to perform projection space filtering on the CPU, using the cache-optimized FFTW library [31]. This is reasonable since the FFT is a relatively sequential algorithm which involves many non-GPU-friendly operations, such as sorting, indexing and bit-wise calculations. Although a number of GPU-based FFT implementations are available [32][33] significant speedups are obtained only for 2D and 1D FFTs in conjunction with sufficiently large arrays (over 10k elements) -- larger than those typically encountered in CT applications. This also resonates well with our declared goal to integrate the CPU as part of our computing pipeline, in order to make optimal use of all available resources. For synchronizing the GPU calculation and the data transfers from CPU to GPU, we take advantage of the Pixel Buffer Objects (PBO) and Vertex Buffer Objects (VBO). Another important issue is the precision used within the pipeline. While the projection data acquired by the detector typically have a dynamic range within 12-16 bits, the filtering potentially widens this range, and therefore maintaining full 32-bit floating point precision in later pipeline stages is most appropriate in clinical settings.

6.2 Backprojection

We represent the target volume as an axis-aligned stack of 2D textures, since a single 3D texture does not support an efficient update mechanism, as mentioned before. As illustrated in Figure a, the CT pipeline begins with generating a series of quadrilaterals P_i (called *proxy polygons*) which define the location and spatial extent of each volume slice as

well as corresponding 2D textures T_i that contains voxel values to be reconstructed (initialized to zero). Then from one or a set of specific projection angles, for each such slice P_i, viewing its host polygon face-on in orthographic viewing mode produces the fragments that relate to the slice voxels (solid arrows in Fig. 5b). The vertex shader then performs the mapping of these fragments into the perspective coordinate space of the back-projected image (the detector space). This is often referred to as *projective-textures mapping* (dashed arrows in Fig. 5b). Using the coordinate producedby the rasterizer, an interpolation of this image at the mapped location produces the fragment value, which is then added/written to the corresponding output texture T_i representing the slice. These operations represent one rendering cycle (pass) on the GPU. New passes will be initialized and executed repeatedly until every volume slice is processed, from every projection angle.

In practical applications, the detector plane may not always be oriented perfectly collinear with the rotation axis for all angles. This may be due to (measurable) gantry instabilities, or it may be intended, in order to implement specific acquisition protocols on non-circular orbits. In order to describe a generalized mapping from volume space into perspective detector space, for a given angle ϕ, we first express the three orthogonal axes of the source coordinate system (X_s, Y_s, Z_s) (see Fig. 2) as unit vectors $\boldsymbol{u}$, $\boldsymbol{v}$, and $\boldsymbol{n}$, and the source position as vector $\boldsymbol{s}$ (note, (X_s, Y_s, Z_s) are collinear to (X_ϕ, Y_ϕ, Z_ϕ), and all of these vectors are originally described in volume space). The full coordinate transformation from volume space to detector space can then be decomposed and expressed by a series of matrices, as shown in Equation (4).

$$S \otimes T \otimes P \otimes M \otimes \vec{v} = \vec{v}_h$$

$$\begin{bmatrix} \frac{w}{2} & 0 & 0 & 0 \\ 0 & \frac{h}{2} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 1.0 \\ 0 & 1 & 0 & 1.0 \\ 0 & 0 & 1 & 1.0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{2n}{w} & 0 & 0 & 0 \\ 0 & \frac{2n}{h} & 0 & 0 \\ 0 & 0 & \frac{f+n}{n-f} & \frac{2fn}{n-f} \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} u_x & u_y & u_z & -\vec{u}\cdot\vec{s} \\ v_x & v_y & v_z & -\vec{v}\cdot\vec{s} \\ n_x & n_y & n_z & -\vec{n}\cdot\vec{s} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_v \\ y_v \\ z_v \\ 1 \end{bmatrix} = \begin{bmatrix} x_h \\ y_h \\ z_h \\ w_h \end{bmatrix} \qquad (4)$$

$$n = \frac{h}{2\tan(\theta/2)}, f = \text{constant} \qquad P_\varphi(Z,Y) = (\frac{x_h}{w_h}, \frac{y_h}{w_h})$$

Representing a voxel coordinate (x_v, y_v, z_v) as a 4D (homogenous) vector, we generate a 4x4 model-view matrix $\boldsymbol{M}$ defined by $\boldsymbol{s}$ and ($\boldsymbol{u}$, $\boldsymbol{v}$, $\boldsymbol{n}$). $\boldsymbol{M}$ transforms the voxel coordinates from the volume coordinate system (X_v-Y_v-Z_v) to the source coordinate system (u-v-n). Another 4x4 projection matrix, $\boldsymbol{P}$, determined by the cone angle θ and the detector dimensions w and h implements the subsequent perspective projection. $\boldsymbol{M}$ and $\boldsymbol{P}$ map voxel coordinates into a canonical view space, which is essentially a volume whose Cartesian coordinates are between -1 and 1. Then, translation $\boldsymbol{T}$ and a scaling matrix $\boldsymbol{S}$, determined by the size of the detector (w and h) produce the homogeneous coordinates in detector space. To compensate for the perspective distortion effects, a division by the 4th component is then required to derive the correct pixel coordinate (for more details on this type of mapping see [11][34]).

On the GPU, the above process (except the final division) is performed on the 4 vertices of the proxy polygons in the vertex processing stage to generate their homogenous coordinates. The affine properties of the matrix operation then allow the fast hardware rasterization engine, via linearly interpolation of these coordinates, to produce the correct coordinates of the fragments in between. Once the fragments are generated, the fragment program performs the final perspective division, producing the detector coordinates needed for the sampling of the (filtered) projection image, which are streamed in as textures from the CPU. The sampling position usually does not coincide with the detector pixels and a sampling kernel needs to be applied to produce final values (see Fig. 3b). Here, the method employed for this sampling is important. We have found that, in conjunction with projections where the resolution exceeds that of the reconstructed volume, the less costly nearest-neighbor sampling produces satisfactory result (having actual high-resolution data is superior to bilinear filtering of matched-resolution data). However, we have also added support for bilinear interpolation, which provides better anti-aliasing when the size of the projections is on the same order than that of the volume slices.

Once the fragment values have been generated, we accumulate them into the corresponding slice voxels. The above procedure is repeated for each volume slice until the entire volume is updated. We then move to the next filtered projection generated by the scanner. The projection-processing iteration over all slices is indicated by the solid arrow in Fig. 5.

6.3 Volume-space weighting

We perform the weighting along with the backprojection within the same vertex and fragment shader program. According to the FDK algorithm shown in equation (1), the extra per-voxel weighting w_v is defined as $d^2/(d+\boldsymbol{r}\cdot x_\varphi)^2$, where d is the distance from the source to the rotation center. Here d is a constant with respect to a certain view angle, while the denominator can be interpreted as the distance from the voxel to the plane that contains the source and is parallel to the detector. This factor can be easily computed by using the center ray vector $\boldsymbol{u}$ and the source position $\boldsymbol{s}$. We calculate the denominator for four vertices in the vertex shader and again take advantage of the rasterization engine to linearly interpolate the values in between. The final weight w_v is then calculated in the fragment shader by an extra division and square operation to multiply with the sampled value.

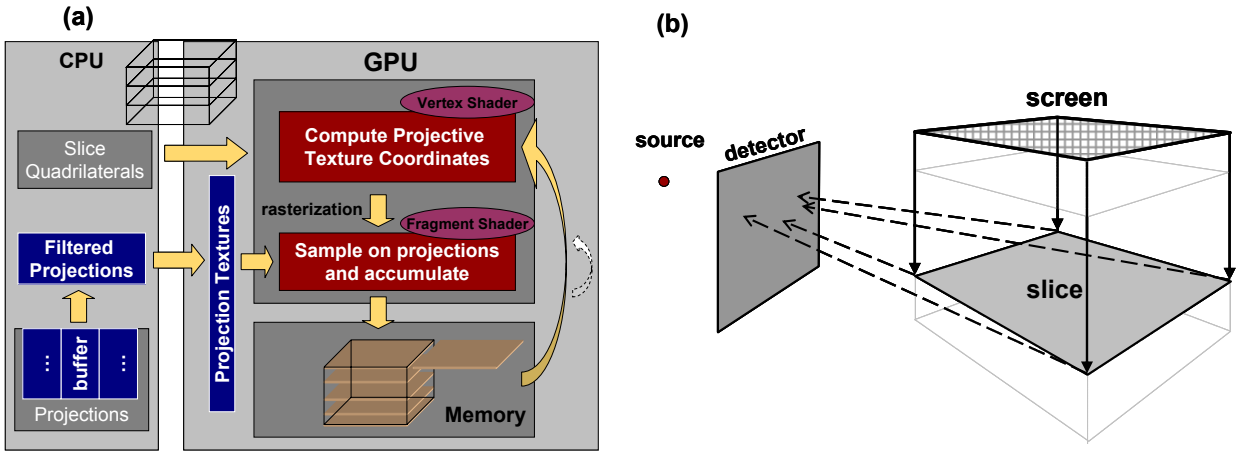

Figure 5: Streaming-CT: (a) pipeline (b) backprojection mapping

6.4 RGBA packing

Originally designed to process graphics primitives, the GPU architecture is capable of efficiently processing 4-component vectors containing red, green, blue and alpha properties of such a primitive. This feature offers an opportunity to obtaining a second level of parallelism, that is, channel parallelism. In our CT reconstruction acceleration, when projections are acquired under or rebinned into parallel-beam geometry and have the same resolution than the object grid, adjacent volume slices share similar sampling patterns on the detector, which is independent of the rotation axis Y. Therefore, every four neighboring projection rows and corresponding volume slices can be packed into a texture of 4 channels and all the transformation and sampling can be implemented concurrently on the GPU. This method reduces the number of rendering passes by four, and in practice yields a 2-3 fold speedup.

7. RESULTS

We used a 3D version of the Shepp-Logan phantom to test our framework. All experiments were performed on a 2.2GHz dual-core Athlon PC with 1GB RAM, equipped with dual Nvidia Geforce 7800 GTX cards. Each GPU has 256 MB on-board memory, running on driver version 91.31 (released June, 2006). The phantom projections were calculated analytically. All projections were acquired on a full circular orbit at a 15° cone angle.

7.1 Reconstruction quality

For the following experiments, we used 360 projections of size 512^2 each to reconstruct a 512^3 volume. Fig. 6 shows slices from the reconstructed 3D Shepp-Logan phantom (at the original 0.5% contrast) from both parallel-beam (0°) and cone-beam (15°) projections, each obtained with our GPU framework as well as a traditional high-quality CPU implementation. We also compared two interpolation schemes, box (nearest neighbor) and bilinear. A Shepp-Logan filter was used for pre-filtering. The shifted profile in the cone-beam images results from the non-exact FDK reconstruction of the off-center slices. We found that both interpolation kernels, bilinear and box, yield excellent and, by visual inspection, nearly identical results. We observe that the box filter tends to produce somewhat sharper, but also noisier images. These differences, however, can only be discerned when comparing the values on an intensity profile, such as the one across the small tumors in the bottom.

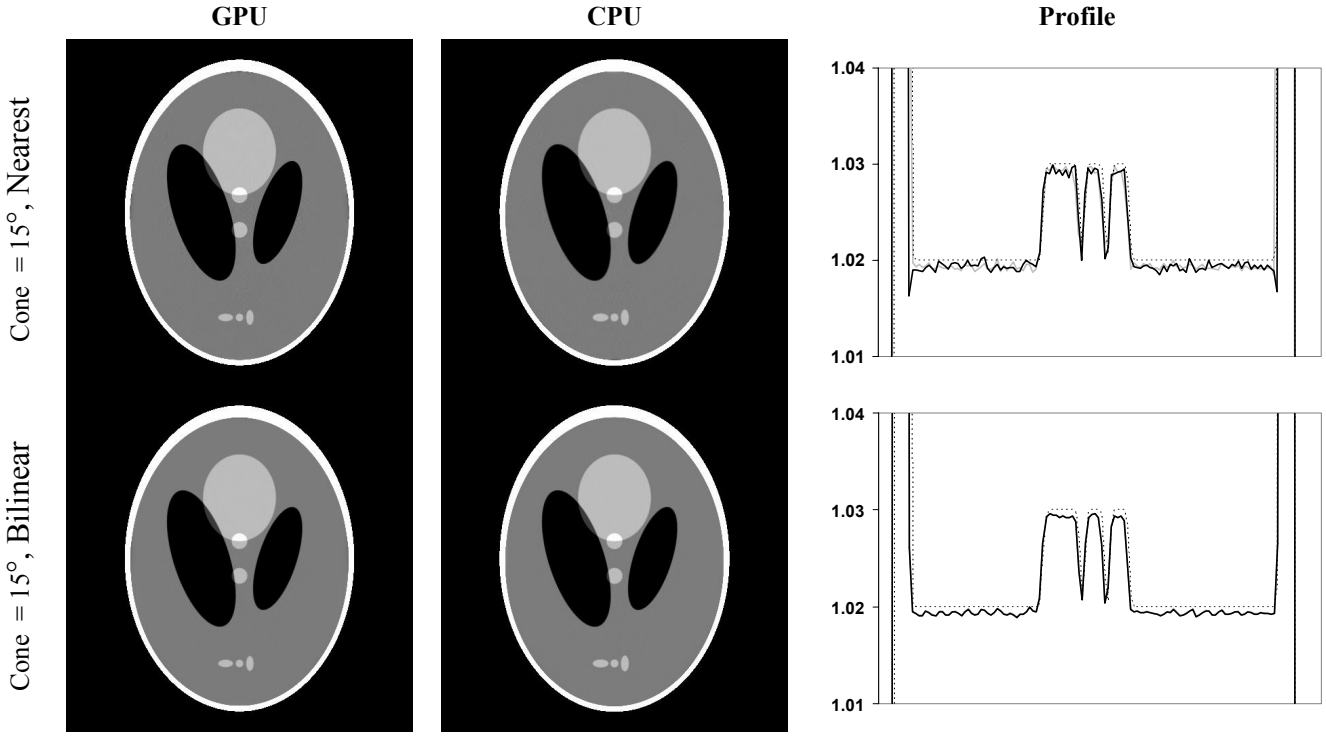

Figure 6. A slice of the 3D Shepp-Logan phantom, reconstructed with our Rapid-CT GPU-based framework (first column) and with a traditional CPU-based implementation (middle column). A windowed density range of [1.0, 1.04] is shown. The right column shows the line profiles across the three tumors near the bottom of the phantom (dashed lines: ground truth; solid gray lines: CPU results; solid black lines: GPU results). We observe that the GPU reconstructions are essentially identical to those computed on the CPU and that they represent the original phantom well. The bilinear filter yields slightly smoother profiles than the box filter, but the reconstruction quality does not suffer significantly when using nearest-neighbor interpolation.

7.2 Reconstruction performance

Table 1 compares the overall performance of our full floating-point Rapid-CT framework with the other high-performance solutions mentioned in Section 2. At the time of publication, we did not have a GeForce 8800 GTX available to us, but using the common benchmark results, one can reasonably expect an increase in performance of factor 2.6 (a factor 1.3 for an upgrade to the GeForce 7900 GTX and factor 2 for a subsequent upgrade to the 8800 GTX). We also list the performance figures obtained with the other approaches enumerated in Section 3, all scaled to our problem size. Finally, we also list the performance of a commercial CPU solution.

	Hardware platform	Time
Rapid-CT (Xu and Mueller)	Single GPU 7800 GTX	19.2 s
	Dual GPU 7800 GTX	10 s
	Single GPU 8800 GTX (est.)	7.4 s
	Dual GPU 8800 GTX (est.)	3.8 s
Exxim Computing Corp.	Dual Pentium 2.4GHz	100 s
Goddard and Trepanier Leeser et al., Li et al	FPGA	33.5 - 38.7 s
Kachelrieß et al.	Cell BE	8 s
	Dual Cell BE	4 s

Table 1. Performance timings for various high-performance CT reconstruction solutions.

8. DISCUSSION AND CONCLUSIONS

The visual results indicate that GPUs can succeed in producing reconstructions at excellent quality. The timing results, on the other hand, are very interesting. We immediately see that GPUs are much more efficient than CPUs (an order of magnitude), which comes at no surprise. They also seem to be more efficient than FPGA solutions, but these works have been completed 2-5 years ago, and it is not clear how much FPGA technology has improved. These kinds of approaches may very well be competitive now. On the other hand, it is interesting to see that all FPGA solutions offer about the same performance, even though they are 3 years apart (in publication date). This may or may not indicate the performance growth patterns. It is also remarkable that the Cell processor solution and the GPU solution have relatively similar timings. This may be expected as they both are incarnations of the streaming paradigm. These similarities in a way reflect the current trends in PC hardware: on one side there are the GPU manufacturers (AMD/ATI, Nvidia) who bet on generalizations of their GPU architecture, and on the other side are the CPU manufacturers (IBM) who bet on a tight and parallel coupling of their traditional CPUs. It is not clear at this point which of the two strategies will win in the end (or if they will converge), but a strong determinant will be market penetration and consumer investment. At the current time, a GPU-based solution looks a lot more cost-effective than a Cell-based solution. But this may change once reasonably-priced Cell-boards are available on the mass market and a few hit computer games are designed to run on them. Perhaps, in order to produce research with lasting value, research should be primarily focused on exploiting the SIMD paradigm for acceleration, and only secondarily on exploiting the short-term advantages a certain architecture currently offers. This was true for GPU acceleration as a whole, and it seems to be true also in this new market setting.

ACKNOWLDGEMENTS

This work was partially funded by NIH grant R21 EB004099-01 and Agard Lab, Department of Biochemistry and Biophysics, University of California at San Francisco.

REFERENCES

1. Z. Fan, F. Qiu, A. Kaufman, S. Yoakum-Stover, "GPU Cluster for High Performance Computing," *ACM/IEEE Supercomputing Conference*, page 47, Nov. 2004
2. http://www.umiacs.umd.edu/research/GPU
3. http://folding.stanford.edu/
4. N. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha, "Fast computation of database operations using graphics processors," *Proc. 2004 ACM SIGMOD'04*, pp. 215-226, 2004.
5. A. Sud, M. A. Otaduy and D. Manocha, "DiFi: Fast 3D Distance Field Computation Using Graphics Hardware," *Computer Graphics Forum,* 23(3), pp. 557-566, 2004
6. N. Govindaraju, J. Gray, R. Kumar, and D. Manocha, "GPUTeraSort: High Performance Graphics Coprocessor Sorting for Large Database Management," *ACM SIGMOD*, 2006.
7. F. Xu and K. Mueller, "Accelerating popular tomographic reconstruction algorithms on commodity PC graphics hardware," *IEEE Trans. Nuclear Science*, 52(3), pp. 654-663, 2005.
8. K. Mueller and F. Xu, "Practical considerations for GPU-accelerated CT," *IEEE Symp. Biomedical Imaging (ISBI'06)*, pp. 1184-1187, 2006.

9. B. Cabral, N. Cam and J. Foran, "Accelerated volume rendering and tomographic reconstruction using texture mapping hardware," *Symp. Volume Visualization 1994*, pp. 91-98.
10. K. Mueller and R. Yagel, "Rapid 3-D cone-beam reconstruction with the simultaneous algebraic reconstruction technique (SART) using 2-D texture mapping hardware," *IEEE Trans. Medical Img.*, 19(12), pp. 1227–1237 2000.
11. M. Segal, C. Korobkin, R. van Widenfelt, J. Foran and P. Haeberli, "Fast shadows and lighting effects using texture mapping" *Proc. Siggraph*, 249–252, 1992.
12. http://www.techpowerup.com/printreview.php?id=/NVIDIA/G80
13. A. Greb and G. Zachmann, "GPU-ABiSort: optimal parallel sorting on stream architectures," *Parallel and Distributed Processing Symposium IPDPS'06,* pp. 25-29, 2006.
14. http://ati.amd.com/
15. D. Pham, et al. "The design and implementation of a first–generation Cell processor," *IEEE International Solid–State Circuits Conference*, pp. 184–185, Feb. 2005.
16. B. Flachs, et al. "A streaming processing unit for a Cell processor," *IEEE International Solid–State Circuits Conference,* pp. 134–135, Feb. 2005.
17. http://www.mc.com/cell/
18. W. Mark, R. Glanville, K. Akeley, and M. Kilgard, "Cg: a system for programming graphics hardware in a C-like language," *ACM Trans. Graphics,* 22(3), pp. 896-907, 2003.
19. I. Buck, et al., "Brook for GPUs: stream computing on graphics hardware," *ACM Trans. Graphics,* 23(3), pp. 777-786, 2004.
20. K. Chidlow and T. Möller, "Rapid emission tomography reconstruction," *Volume Graphics '03*, pp. 15–26, 2003.
21. H. Hudson and R. Larkin, "Accelerated image reconstruction using ordered subsets of projection data," *IEEE Trans. Medical Imaging*, 13(4) pp. 601–609, 1994.
22. J. Kole and F. Beekman, "Evaluation of accelerated iterative x-ray CT image reconstruction using floating point graphics hardware," *Physics Medicine and Biology*, 5 pp. 875-889, 2006.
23. X. Xue, A. Cheryauka and D. Tubbs, "Acceleration of fluoro-CT reconstruction for a mobile C-Arm on GPU and FPGA hardware: a simulation study," *Proc. SPIE 6142*, pp. 1494-1501, 2006.
24. T. Schiwietz, T. Chang, P. Speier and R. Westermann, "MR image reconstruction using the GPU," *Proc. SPIE 6142*, pp. 1279-1290, 2006.
25. F. Xu and K. Mueller, "GPU-Accelerated D2VR," *Volume Graphics 2006*, pp. 23-30, Boston, MA, August 2006.
26. 26.I. Goddard and M. Trepanier, "High–speed cone–beam reconstruction: An embedded systems approach," *SPIE Medical Imaging Proc.*, vol. 4681, pp. 483–491, 2002.
27. M. Leeser, S. Coric, E. Miller, H. Yu, M. Trepanier, "Parallel–beam backprojection: An FPGA implementation optimized for medical imaging," *Proc. Tenth Int. Symposium on FPGA,* Monterey, CA, pp. 217–226, Feb. 2002.
28. J. Li, C. Papachristou, and R. Shekhar, "An FPGA-based computing platform for real-time 3D medical imaging and its application to cone-beam CT reconstruction," *J. Imaging Science and Technology,* 49(3), pp. 237-245, 2005.
29. L. Feldkamp, L Davis, and J. Kress, "Practical cone beam algorithm," *J. Optical Society America A,* 1(6), pp. 612–619, 1994.
30. M. Kachelrieß, M. Knaup, and O. Bockenbach, "Hyperfast Perspective Cone–Beam Backprojection," *IEEE Medical Imaging Conference 2006*.
31. M. Frigo and S. Johnson, "The design and implementation of FFTW3 Proc," *Proc. IEEE,* 93(2) pp. 216–231, 2005.
32. N. Govindaraju, S. Larsen, J. Gray, and D. Manocha, "A memory model for scientific algorithms on graphics processors," *UNC Technical Report* available at http://gamma.cs.unc.edu/GPUFFTW, 2006.
33. T. Sumanaweera T and D. Liu, "Medical image reconstruction with the FFT," GPU Gems II, Addison Wesley, pp. 765-784, 2005.
34. J. Foley, A. van Dam, S. Feiner, and J. Hughes, *Computer Graphics: Principles and Practice*. New York: Addi-son-Wesley, 1990.