ULTRA-FAST 3D FILTERED BACKPROJECTION ON COMMODITY GRAPHICS HARDWARE

Fang Xu and Klaus Mueller

Center for Visual Computing, Computer Science Department, Stony Brook University, NY, USA

ABSTRACT

Recent efforts in cone-beam scanner technology have focused on developing interactive scanning capabilities, for example, to enable image-guided surgical interventions or real-time diagnosis with time-varying data. However, apart from a fast scanner these applications also require a fast reconstruction algorithm to match. The Filtered Backprojection algorithm devised by Feldkamp, Davis, and Kress is the most widely used algorithm for 3D reconstruction from cone-beam projections, and it is the algorithm with the lowest complexity. Yet, pure software implementations have difficulties to process the data at the speeds required for real-time scanning. One option is to utilize expensive and rare custom boards for this purpose. We describe an alternative solution, which is inexpensive, uses readily available PC graphics hardware boards, and provides the desired performance at the quality required.

1 INTRODUCTION

The Filtered Backprojection algorithm proposed by Feldkamp, Davis, and Kress (FDK) [6] is by far the most popular cone-beam reconstruction algorithm. It has been employed by a great number of researchers in conjunction with various types of cone-beam scanners (see e.g., [5]). The FDK algorithm first filters the projection data and then accomplishes the reconstruction by back-projecting the filtered projections. These backprojection operation are also part of algorithms that perform exact 3D reconstruction via the inverse Radon transform [4][10]. In fact, the FDK algorithm is a special (non-exact) case of the inverse Radon transform for a single circular source/detector orbit. The backprojection procedure has the highest complexity with $O(N^4)$ complexity, making it the most time-consuming part of any of these algorithms.

Recent efforts in cone-beam scanner technology have sought to develop interactive scanning capabilities, for example, to enable image-guided surgical interventions or real-time diagnosis with time-varying data, such as the beating heart. However, the speed at which 3D reconstruction can be achieved with pure-software implementations is limited by the performance of general-purpose CPUs. For example, the reconstruction of a 256^3 volume from 256 projections can take over 10 minutes, even with simple nearest neighbor interpolation. To overcome these limitations, custom chips and boards (an ASIC chip by Terarecon Inc. and an FPGA board by Mercury Computer Systems Inc.) have been introduced in recent years. Although impressive reconstruction speeds in the range of seconds can be achieved, the high cost of these boards somewhat limits their accessibility to both researchers and many medical institutes. In addition, once conceived, these implementations offer little flexibility to accommodate algorithmic updates. Thus, while these boards are certainly an economically sound solution for the incorporation of proven technology into high-priced scanners, a more main-stream solution is also desirable for more experimental and research settings.

In 1994, Cabral, Cam and Foran [2] performed accelerated 3D reconstruction utilizing the hardwired, multi-pipelined texture mapping hardware resident in mid-range SGI workstations. Mueller and Yagel [12] implemented the Simultaneous Algebraic Reconstruction Technique (SART) [1] algorithm on this platform. This hardware, however, had several shortcomings: (1) it was expensive (the cost of these workstations was over $20k), (2) only integer arithmetic was available, limiting the accuracy that could be achieved, and (3) the hardware was not programmable, limiting the types of operations that could be performed. To overcome shortcomings (1) and (2), some portions of the backprojections (accumulations, divisions, etc) had to be performed on the CPU, slowing performance due to the small bandwidth at the CPU-graphics interface.

The emergence of high-end PC-based graphics boards, made economical by the ever-growing demands of computer games, has brought accelerated texture mapping to the consumer market and helped overcome shortcoming (1) of the previous approaches. Chidlow and Möller [3] used the NVidia GeForce4 board for 3D emission tomography reconstruction from a stack of fan beam data with the OS-EM algorithm [8]. However, a considerable portion of the computations still had to be performed on the CPU since the graphics hardware only provided 8-bit arithmetic. This limited performance a great deal. To enhance precision, they extended a scheme introduced by Mueller and Yagel [12] in which the data words were split among the RGBA color channels. This allowed the projection/backprojection operations to be performed on the board, while the end result needed to be assembled on the CPU.

The latest generations of graphics cards (GPUs) from NVidia and ATI (the NVidia GeForce FX and the ATI Radeon) finally overcome all three of the previous shortcomings: They are fully programmable, offer special ALU pipelines with floating point arithmetic, and yet cost less than $500. In another paper [15], we demonstrated that the floating point pipelines enabled speedups at the order of 1-2 magnitudes for a variety of reconstruction algorithms, such as SART [1], OS-EM [8], and FDK. However, we recently found that these floating point pipelines are 4-5 slower than the hardwired 8-bit texture mapping facilities, leading to a less-than-expected GPU performance of the FDK algorithm in particular. To cope, we explore a novel scheme that judiciously splits calculations among the 8-bit and the floating point hardware for optimal performance, but without significantly compromising reconstruction quality. We will start with some background information, then describe our implementation, and finish with results and conclusions.

2 BACKGROUND

2.1 The FDK algorithm

The FDK projection geometry is illustrated in Fig. 1, while the algorithm is written as follows:

for each projection $proj_k$
  // perform filtering
  weight pixels by a/b
  ramp-filter each column (y_d direction)
  // perform backprojection
  for each grid voxel v_j
    project v_j onto image along cone-beam rays
    interpolate voxel update dv_j
    weight dv_j by depth factor c_j: $dv_j = dv_j \cdot c_j$
    add result to grid voxel: $v_j = v_j + dv_j$

where the depth weighting factor c_j is:

$$c_j = \frac{a^2}{\left(a + \sqrt{v_{jy}^2 + v_{jz}^2}\cos(\varphi - \varphi_k)\right)^2} \quad (1)$$

This depth factor is independent of the (axial) x-coordinate.

2.2 Graphics hardware fundamentals

Graphics objects are typically composed of polygon meshes, where additional surface detail can be modeled by affixing (or *mapping*) images (or *textures*) of the desired detail onto the polygons during the rendering phase. Texture mapping is an efficient way to provide intricate surface detail without increasing an object's polygon count, and graphics hardware is highly optimized to perform texture mapping very fast, even under perspective distortion [7]. There are two main stages in a graphics pipeline: the *geometry processing stage* and the *polygon rasterization stage*. In the former, the geometric information, i.e., the polygon vertex coordinates, are transformed to determine their screen space coordinates. Then, in the rasterization stage these projected vertices are connected to form the (projected) polygon, whose content is filled (or *rasterized*), combining colors interpolated from the polygon's vertex attributes and from the mapped texture. The pixels so generated are called screen *fragments*. Graphics processors gain their high polygon throughput rates (close to 400M/s) by providing highly parallel, hardwired logic for both geometric processing and rasterization. However, while (parts of) these computations are performed at floating point precision internally, the precision of the output fragments (i.e., their RGBA color and opacity values) has always been 8 bit (12 bit on the SGIs). As was stated above, this limited the utility of the hardware for CT applications. Fortunately, the latest hardware has added a second data path for both rasterization and geometric processing – a set of fully programmable floating point pipelines, that, similar to the hardwired units, operate in SIMD (same instruction, multiple data) mode. It turns out, however, that, while still much faster than a CPU, these floating point pipelines are about 4-5 time slower than the rasterizers (as well as the geometry processors), which, in part, is due to the higher memory bandwidth required for the 4-byte floating point data. It seems therefore desirable to distribute the workload to the hardwired rasterizers as much as possible, whenever 8-bit precision can be tolerated.

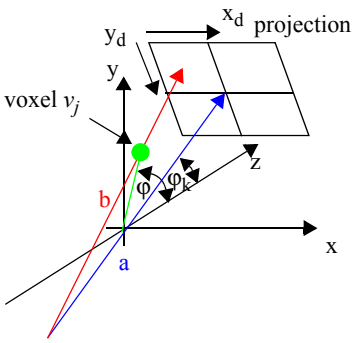

Figure 1: FDK projection geometry.

GPUs, such as the NVidia FX 5900 or ATI 9800 are considered *stream processors* [9]. Each of the 8 parallel pipelines can accept up to 16 different input vectors or *streams* (stored in 2D textures), perform a computation on them, and produce an output stream, which is also stored into a 2D texture. GPUs are considerably faster than CPUs for stream-computations, due to their high memory bandwidth, and their highly parallel and pipelined SIMD architecture. Finally, additional task-parallelism can be gained from performing simultaneous calculations in the RGBA channels of the pipelines. Thus, any algorithm that features long loops of independent operations can fully benefit from the high-performance stream architecture that GPUs can offer, and the FDK reconstruction method is such an algorithm.

3 IMPLEMENTATION

3.1 General algorithm

To illustrate our implementations, we rewrite the FDK method in vector notation:

$$V = \sum_{\varphi \in S} B(F(P_\varphi)) \quad (2)$$

where F is the filtering operator, B is the backprojection operator, V is the reconstructed volume, S is the number of projections in the set, and P_φ is the projection image taken at angle φ. We currently perform F on the CPU since it is a less costly $O(N^2)$ algorithm per projection, and we intend to keep all computational units busy. Once filtered (and scaled to 8 bit), the projections are streamed into the GPU. The overhead is minimal since the loading of the GPU's texture memory can occur simultaneously with GPU calculations.

We have mentioned before that the GPU's hardwired 8-bit rasterizers are significantly faster than the GPU's floating point units. Thus we would like to use them as much as possible. The most compute intensive portion of the remaining part of equation (2) is the backprojection, since it requires bilinear interpolation. In many applications, the projection images obtained from the scanner are 8-bit (we shall address the case of 12-bit and 16-bit images later). We do not require the interpolation to return results at significantly higher accuracy (note that this may be different for iterative algorithms). Thus there is a real potential for using the hardwired rasterizers for the backprojection. The accumulation, however, must be executed in floating point precision since the range of values is likely to overflow 8 bits.

Our general algorithm is illustrated in Fig. 2. To avoid excessive context switching between 8-bit projections and floating point summing, we process the volume slice-by-slice. We start by initializing a large 2D texture, with S 2D tiles of size N^2 each. Using the backprojector described later, we backproject each image with the viewing geometry set appropriately to the current volume slice and

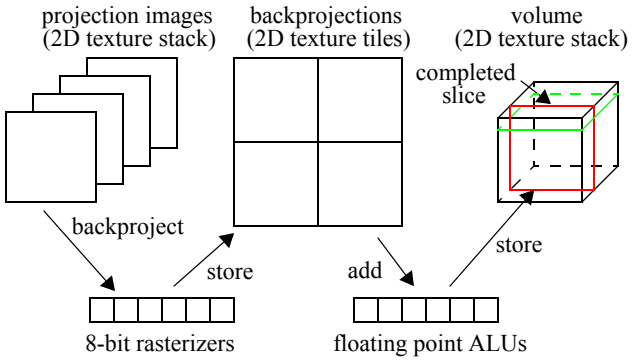

Figure 2: FDK hardware implementation overview. All steps to calculate one volume slice (shown in red) are shown.

the projection angle φ. We store each backprojection result in the proper tile. Once all S projections have been backprojected, we sum the tiles as a vector add in the floating point pipelines. Currently, GPUs allow us to add 10 vectors in one pass. Thus, if S=128, we can complete the backprojection of one volume slice in 128 8-bit passes, one context switch, 13 floating-point passes, and another context switch to proceed to the next slice.

The volume slices are represented as 2D textures mounted onto a stack of parallel polygons (see Fig. 2). Although it is easy to forward project such a volume under perspective (cone-beam) geometry, it is much more difficult to achieve backprojection with this direct approach, since now the "screen" is formed by the polygon and the line of slight is not perpendicular to this screen (Fig. 3). We shall now describe two methods that achieve this backprojection.

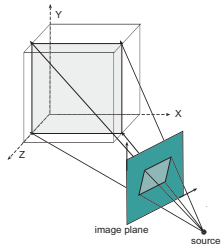

Figure 3: Perspective projection.

3.2 Backprojection via projective textures

The first method is the "inverse" of the forward projection of Fig. 3, but uses an indirect projection mode called *projective textures* [14]. It works similar to a slide projector (see Fig. 4). This method is described in detail in [12]. Briefly, the backprojected image forms the "slide", which is perspectively projected onto the "screen" formed by a polygon that is placed at the location of the volume slice to be updated. The "slide projection" is then "viewed" in parallel projection mode on the screen (i.e., it is rasterized into the framebuffer). Here, the perspective transform is given by the

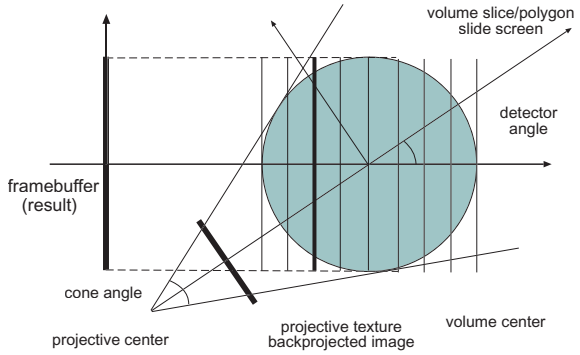

Figure 4: Backprojection with projective textures.

viewing geometry at which the projection was originally obtained from the scanner. A downside of the method is that two texture stacks are needed, one for each major projection direction, and that these texture stacks have to be merged at the end.

3.3 Backprojection via texture spreading

In the projective texture approach an entire 2D texture must be accessed/streamed through the pipeline during each backprojection (compare Fig. 3). This can cause bottlenecks when the memory bandwidth is less than the compute bandwidth. An alternative backprojection approach is *texture spreading,* which minimizes the required memory bandwidth. Instead of 2D projective texture sampling, this approach spreads a 1D texture across the frame buffer and updates the horizontal slices (the y-axis stack, green slice in Fig. 2) instead of the vertical slices (the x-axis and z-axis stacks, red slice in Fig. 2). This also eliminates the need for the texture stack merge that arises with projective textures.

In a parallel projection geometry, each volume slice along the vertical axis corresponds to a single row of the projection data. The resulting spreading operation can be implemented by mapping the 1D texture line endpoints to the near and far end of the volumes slice's polygon (Fig. 5a). Off-axis projections can be implemented by rotating the slice polygon first and then projecting it to the screen, texture-mapped as just described (Fig. 5b). This will only require the fetch of a single texture line from GPU memory, which optimizes the required memory bandwidth. We extend this basic approach, first introduced in [3], to cone-beam backprojections. It can be achieved by a cone- and fan-angle conformant mapping of the slice polygon vertices into the texture space of the backprojected image (see Fig. 5c). Although now perhaps 4-5 lines of the texture needs to be fetched per slice (Fig. 5d), depending on the volume size and the cone-angle, the required bandwidth is only a

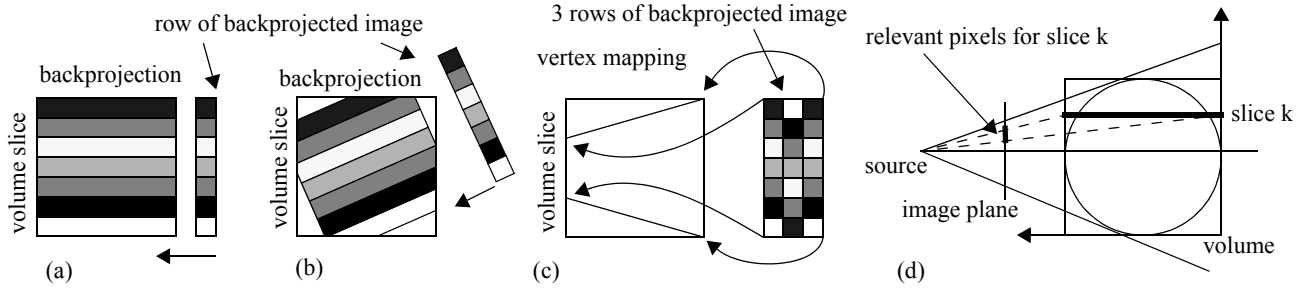

Figure 5: Backprojection with texture spreading: (a) backprojection result for parallel beam (spreaded row); (b) backprojection for parallel beam at an off-angle position; (c) general backprojection for cone-beam geometry; (d) side view for cone-beam geometry.

small fraction of that of the projective texture method.

3.4 FDK depth weighting

We enable the depth-weighting of the FDK algorithm (equation (1)) using 2D lookup textures (called *dependent textures*), one for each principal projection orientation angle φ_k. Each dependent texture is indexed by the slice voxel's y and z-coordinates (the x-coordinate is not relevant) and multiplied by the voxel value during the accumulation step. We currently compute this map in software as a pre-process, but a hardware implementation is possible..

4 RESULTS

Fig. 6 presents reconstructions we have obtained with our implementation, while Table 1 presents the timings for these. The numbers do not include the time required for filtering. We observe that the full floating point GPU implementation of the backprojection can achieve speedups of 7.5 with excellent image quality, while the approach using 8-bit rasterization is somewhat more noisy, but gives a speedup of 37 (both compared to a fairly optimized software implementation). We did not notice qualitative nor runtime differences between the projective and texture spread approach. This, however, could change over time, should hardware manufacturers improve memory and computational bandwidths at different rates, driven by computer game market demands

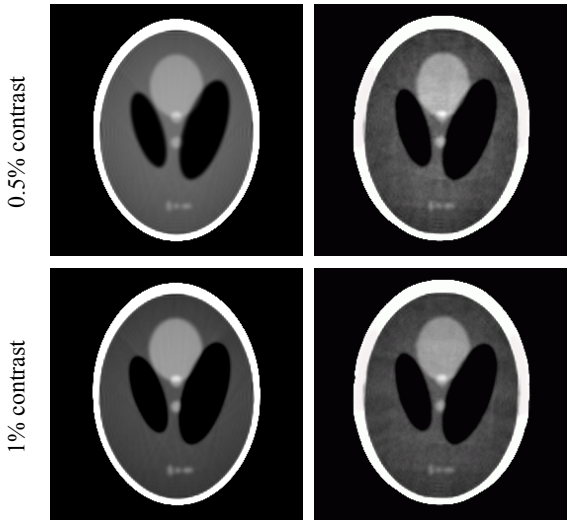

Figure 6: Slice of the reconstructed Shepp-Logan phantom for various hardware implementations and contrast settings.

Implementation	Time (128^3 / 160)	Time (256^3 / 320)
Software (CPU)	1 min	16 min
GPU - full floating point	8 s (7.5)	2 min
GPU - 8-bit projective textures	1.6 s (37)	25 s
GPU - 8-bit texture spreading	1.6 s (37)	25 s

Table 1: Timning results for the various tested FDK implementations. The timings in the center column are for a 128^3 volume and 160 projections, while the timings in the right-most column are for a 256^3 volume and 320 projections. Speedups (over CPU) are given in parentheses.

5 CONCLUSIONS

The impressive reconstruction performance and quality that can be reached with the commodity GPU implementations seem sufficient for many interactive clinical applications, such as image-guided surgical interventions. The performance is near real-time for smaller volumes, facilitating region of interest CT, and less than a minute for full-scale CT datasets. Current work focuses on the reconstruction of large volumes, with projections of 12-bit precision and greater. For this, we intend to use the channel splitting technique described in [12]. Since GPU performance has so far doubled every 6 months (i.e., triple of Moore's law), we expect that the gap between CPU and GPU approaches will widen even further in the near future.

REFERENCES

[1] A. Andersen and A. Kak, "Simultaneous Algebraic Reconstruction Technique (SART): a superior implementation of the ART algorithm," *Ultrason. Img.*, vol. 6, pp. 81-94, 1984.

[2] B. Cabral, N. Cam, J. Foran, "Accelerated volume rendering and tomographic reconstruction using texture mapping hardware," *1994 Symp. on Volume Visualization*, pp. 91-98, 1994.

[3] K. Chidlow and T. Möller, "Rapid emission volume reconstruction," *Proc. Volume Graphics Workshop* 2003, pp. 15-26.

[4] M. Defrise, R. Clack, "A cone-beam reconstruction algorithm using shift-variant filtering and cone-beam backprojection," *IEEE Trans. Med. Img.*, vol. 13, no. 1, pp. 186-195, 1994.

[5] R. Fahrig, A. Fox, S. Lownie, D. Holdsworth, "Use of a C-arm system to generate true 3-D computed rotational angiograms: Preliminary in vitro and in vivo results," *Am. J. Neuroradiol.* 18, pp. 1507-1514, 1997.

[6] L. Feldkamp, L. Davis, and J. Kress, "Practical cone beam algorithm," *J. Opt. Soc. Am.*, pp. 612-619, 1984.

[7] J. D. Foley, A. van Dam, S. K. Feiner, J. F. Hughes, *Computer Graphics: Principles and Practice*. Addison-Wesley, 1990.

[8] H. Hudson, R. Larkin, "Accelerated Image Reconstruction Using Ordered Subsets of Projection Data," *IEEE Trans. Medical Imaging*, vol. 13, pp. 601-609, 1994.

[9] U. Kapasi, W. Dally, B. Khailany, J. Ahn, P. Mattson, and J. Owens, "Programmable stream processors," *IEEE Computer*, vol. 36, no. 8, pp. 54-62, 2003.

[10] H. Kudo and T. Saito, "Derivation and implementation of a cone-beam reconstruction algorithm for non-planar orbits" *IEEE Trans. Med. Imag.*, vol. 13, no. 1, pp. 196-211, 1994.

[11] W. Mark, S. Glanville, and K. Akeley, "CG: A system for programming graphics hardware in a C-like language," *Proc. SIGGRAPH'03*, pp. 896-907, 2003.

[12] K. Mueller, R. Yagel, "Rapid 3D cone-beam reconstruction with SART by using texture mapping hardware," *IEEE Trans. on Medical Imaging,* vol. 19, no. 12, pp. 1227-1237, 2000.

[13] J. Neider, T. Davis and M. Woo, *The Official Guide to Learning OpenGL*. Addison-Wesley, 1994.

[14] M. Segal, C. Korobkin, R. van Widenfelt, J. Foran, and P. E. Haeberli, "Fast shadows and lighting effects using texture mapping," *SIGGRAPH'92)*, vol. 26, pp. 249-252, 1992.

[15] F. Xu and K. Mueller, "Accelerating popular tomographic reconstruction algorithms on commodity PC graphics hardware," *Trans. Nucl. Sci.*, (in review), 2003.