RI11003 August 9, 2011 Computer Science

IBM Research Report

3D CT Reconstruction on GPU using OpenGL

Monu Kedia and Yogish Sabharwal

IBM Research India
4, Block C, ISID Building, Institutional Area,
Vasant Kunj, New Delhi - 110016. India.

IBM Research Division
Almaden - Austin - Beijing - Delhi - Haifa - T.J. Watson - Tokyo - Zurich

LIMITED DISTRIBUTION NOTICE: This report has been submitted for publication outside of IBM and will probably be copyrighted is accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties). Copies may be requested from IBM T.J. Watson Research Center, Publications, P.O. Box 218, Yorktown Heights, NY 10598 USA (email: reports@us.ibm.com). Some reports are available on the internet at http://domino.watson.ibm.com/library/CyberDig.nsf/home

3D CT Reconstruction on GPU using OpenGL

Monu Kedia and Yogish Sabharwal
{monu.kedia, ysabharwal}@in.ibm.com
IBM Research, New Delhi, INDIA

Contents

1 Abstract **3**

2 General Purpose Computing on GPU (GP-GPU) using OpenGL **3**
2.1 Programming concepts 3

3 3D CT Reconstruction **5**
3.1 Feldkamp and OSC algorithmic details 5
3.2 X86 implementation used 5

4 Implementation details and optimization techniques **5**
4.1 3D CT reconstruction using Feldkamp algorithm 6
4.2 3D CT reconstruction using OSC algorithm 6
4.2.1 Forward projection 6
4.2.2 Backward projection 11

5 Results and Discussions **16**

1 Abstract

Graphics Processing Units (GPUs) has been traditionally designed for graphics applications. There is recent trend in harnessing computational power offered by GPUs for general purpose computation. Graphics hardware and driver do not expose general purpose programming language like C to the programmer. So, mapping a given computational task to GPUs requires expressing the task in terms of programming model exposed by GPUs. There has been recent efforts to expose languages like CUDA (Compute Unified Device Architecture) [12] which are more tuned towards harnessing GPUs for general purpose computation. However, traditional graphics oriented APIs like OpenGL [11] for general purpose computation using GPUs remain popular for portability reasons. Using APIs like OpenGL for doing general purpose compution on GPUs is challenging and necessitates exploration of several optimization techniques. This report discusses our experience of implementing two algorithms for 3D CT reconstruction on GPUs using OpenGL.

2 General Purpose Computing on GPU (GP-GPU) using OpenGL

2.1 Programming concepts

A typical programming pattern in using OpenGL to carry out general purpose computation is representing structured (array like) read only input data to the program as textures, representing read-only non-structured data as the uniform variable (so called because their value don't change during the rendering), writing a vertex program (also called vertex shader) for per vertex operation, writing a fragment program for per-fragment operation, representing the output data as the write-only texture attached to the frame-buffer, drawing (rendering) a quadrilateral to invoke the computation, reading back the texture attached to the frame buffer containing the results. The computation is invoked by rendering the quadrilateral and there can be only one vertex and/or fragment program for a given rendering pass. So, in one rendering pass the model of computation is data parallel processing or similar to stream computing model. However, different rendering pass can have different fragment and vertex shader, so can perform different computation.

- Compute units : OpenGL exposes several hardware features in terms of vertex processors, rasterizer and fragment processors. Some of them are programmable by the application writer, while other has fixed non-programmable functionality. Apart from this there are hardwares few more specific hardware supports available for features like z-culling etc.

  Vertex Processor : The input to the vertex processors are vertex of the geometry we are rendering. From the GP-GPU point of view we can look vertex processor as modifying the per-vertex attributes in a programmable way. For example, vertex processor can modify the vertex position in the way programmer wants. Also, it can attach programmer defined per-vertex attributes.

  Rasterizer : Rasterizer is a hardwire computational resource available in the GPUs. The function of rasterizer is to interpolate the per vertex attributes processed by the vertex processor and make the interpolated values available to the fragment processor. As is clear, a lots of data corresponding to the fragments is generated which is passed to the fragment processor. Rasterizer is very beneficial for the applications which requires values which can be obtained by the interpolation of some boundary values.

  Fragment Processor : Fragment processor performs per-fragment operation and write the processed value to the frame-buffer. It has access to the interpolated values of the vertex attributes and can gather data from multiple textures. The processing is done in 4-way SIMD corresponding to the RGBA color channel. The output of the fragment processor is written to the frame-buffer. The fragment processor naturally exposes the data-parallel or stream computing model.

- GPU Programming model : The OpenGL exposes data parallel programming model for general purpose computation. The vertex processor can process multiple vertices in parallel and similarly, fragment processor can process multiple fragments in parallel. The second level of parallelism comes from the SIMD processing of RGBA data components in fragment processor.

- Data representation and access : The data storage exposed by OopenGL are either read only or write only. Read-write data is not available in the same rendering pass. If we need data feedback (read-write), than multiple passes are needed along with the technique of ping-pong. In this, the output of the previous pass is treated as the read-only texture in the next pass.

- Data transfers : Data needs to be transfered bak and forth between GPU graphics memory and the host. Depending on the application data requirements, this can be a performance bottleneck and hence, reuse of data tried wherever possible.

3 3D CT Reconstruction

This section discuss the 3D CT reconstruction algorithms which we have implemented on GPU. 3D CT reconstruction refers to the technique of reconstructing the scanned object from the experimental projection obtained using source-detector setting. Two algorithms were used for in our work on 3D CT reconstruction :

3.1 Feldkamp and OSC algorithmic details

The details of Feldkamp and OSC algorithms can be looked at [4], [5] and [6] respectively.

3.2 X86 implementation used

The C implementation used for Feldkamp is exactly the same as described in [7]. However, in OSC [1] instead of using the exact length, the forward projection is computed by sampling the volume at unit distance across the ray while backward projection is computed by extrapolation of the difference. The updates are carried out as per the OSC equation.

4 Implementation details and optimization techniques

This section discuss about the implementation details of Feldkamp and OSC algorithm on GPU. We explain the optimizations carried out and how different GPU resources have been utilized to get better performance. The following subsection discuss about the Feldkamp and OSC algorithm for 3D CT reconstruction subsequently.

4.1 3D CT reconstruction using Feldkamp algorithm

The mapping of Feldkamp ([7]) algorithm to GPU is trivial and not challenging due to its data parallel nature. So, we skip its details in the current discussions.

4.2 3D CT reconstruction using OSC algorithm

The pseudo code for our implementation is shown in Algorithm 4.1

Algorithm 4.1 OSC Reconstruction

```
OSCRun ( currVol, expProj, NUM_SUB, MAX_ITER )
begin
   ComputeEntryExitMaps() ;
   InitZeroVoxels() ;
   For iter ← 0 to ( Convergence or MAX_ITER )
      For subset ← 0 to NUM_SUB
         ComputeFP( currVol, fpProj, subset, entryExitMap ) ;
         ComputeBP( currVol, fpProj, expProj, subset ) ;
      End For
   End For
end
```

4.2.1 Forward projection

- *Basic Approach :* The GPU implementation of the forward projection is similar to the algorithm used in C implementation as explained in section 3.2. In order to compute the forward projection (FP) value of a projection point, rays are drawn from the center of the projection point to the source as shown in Figure 1.

  The drawn ray intercept the reconstructed volume with some finite length. The intersection points on the face of the volume are the entry and exit points. It is possible that some of the rays don't intersect the volume, in which case the FP value for them will be zero. The FP value is computed by sampling the intercepted ray at unit distance between the entry and exit points. The FP value is calculated by simply accumulating the intensity values at the sampled points. Since the intensity is known only the discrete

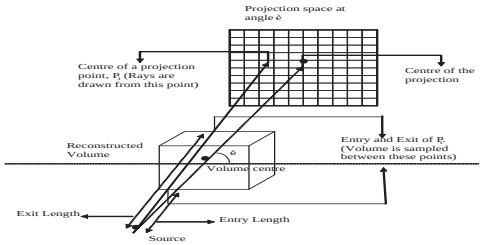

Figure 1: Geometry and rays for forward projection

grid points in the volume space, interpolation techniques are needed to get the value at the sampled points. Two commonly used interpolation techniques are linear interpolation (in this case, trilinear) and nearest neighbor. Trilinear and nearest neighbor have trade offs in terms of image quality and computation time. Trilinear interpolation is computationally expensive but gives better quality image while nearest neighbor is computationally less expensive but image quality may not be as good as trilinear interpolation approach.

Algorithm 4.2 shows the psuedo code in the algorithmic terms along with the comments on the hardware resource utilized.

Algorithm 4.2 OSC Reconstruction : Forward Projection

```
OSCFP ( currVol, fpProj, subset, entryExitMap )
begin
   PackData ( currVol, fpProj ) ;
   For ∀ proj ∈ subset
       Transfer entry-exit map ;
       Set depth buffer ;
       Enable depth test ;
       Set input values to shaders ;
       Render quadrilateral (Invoke computation) ;
       Readback FP results ;
end
```

- *Optimizations* Several optimizations have been incorporated to harness available GPU features. This subsection discuss the details of the incorporated optimizations.

  Precomputed Entry and Exit maps : In order to compute the FP value for a projection point, we need the entry and the exit co-ordinates of the ray inside the reconstructed volume. In 3D two co-ordinates (entry and exit) will require six floating point numbers for each ray. In order to transfer six data point for each projection we need two input RGBA textures. However, we observe that the entry and exit co-ordinates inside the volume for a ray can be computed efficiently by using this entry and exit lengths along with the end point co-ordinates and total length as shown in Figure 2. This optimization reduces the number of data points required to store the entry and exit lengths to two, reducing the number of input texture requirement to one.

  Further, we observe that in the forward projection we don't need to sample the volume from entry to the exit inside the volume, rather we need to sample it from the first non-zero sample value to the last non-zero sample value. It simply means trimming the leading and lagging zeros while sampling the volume along the line. This observation is implemented by editing the entry-exit distance appropriately and as shown in Figure 2. This optimization is most effective when the actual object in the reconstruction volume is concentrated around the center of the volume. Further, due to zero voxel initialization in the backward projection (as discussed later) editing of the entry and exit maps is required only once.

  Texture access optimization by data duplication : The forward projection for the projection points (rays) are computed in the fragment processors. The

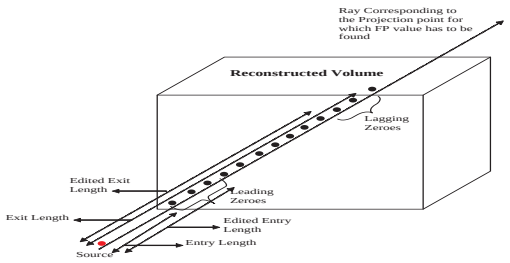

Figure 2: Editing Entry-Exit Maps

key inputs are the current volume and the entry-exit map. As mentioned earlier, sampling of volume is required to compute the forward projection of a ray and this needs interpolation techniques for discrete volume case. The trilinear interpolation requires fetching 8 surrounding texture points and than computing the intensity value as per the interpolation equation. The nearest neighbor technique is cheaper in terms of both memory fetch and computation. Since fragment processors are processing several fragment (rays) in flight, and each such ray may requires several sampling points, fetching data from graphics textures can be a huge bottleneck particularly in the case of tri-linear interpolation. The number of memory fetches can be minimized by duplicating and packing the data as shown in Figure 3. This reduces six memory fetch (for Trilinear interpolation) required to evaluate one sampling point to two.

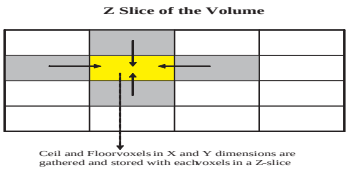

Figure 3: Data packing for optimized memory accesses

It is evident that this data duplication operation can itself become a performance bottleneck if performed on the CPU. In our experiments, we observed this operation is not a bottleneck on lower end GPUs where GPU computation of FP dominates the overall computation time, however on high end GPUs the packing operation itself become a bottleneck if done on CPU. In our implementation, we offloaded this operation also to GPU using separate rendering passes.

Data transfer optimizations : This bullet refers to the data transfered from the CPU memory to the GPU graphics memory. Apart from few constants (like volume size, projection angle etc.) there are two key data structures needed by the GPU device to compute FP. First, the current reconstructed volume which needs to be sampled. Second, the entry-exit map for the projection. For the first case, the entire volume (represented as a 3D texture)

is transfered once and than it is used for computing the FP values for all the projections. The data duplication and packing optimization as explained in previous bullet ensures texture RGBA channels are used to full capacity. For the second case, the length based computation for entry and exit co-ordinate requires only one 2D texture, minimizing the data transfer cost. This transfer is done once for each projection in each iteration.

Z-cull optimization : In FP each of the projection point (or ray) is finally processed as a fragment in the fragment processor of the GPUs. It is clear that if the experimental projection value for a particular projection point is zero than FP value can also be permanently taken as zero. So, for such projection points (rays) we donot need to compute the FP values by sampling the volume. This implies that the fragment corresponding to such projections need not be processed by the fragment processor. The depth test which is performed by the modern GPUs in hardware is very much suitable to implement this optimization. The key idea here is that the depth component of the incoming fragments are tested with the depth value given in an input depth buffer according to a comparison function. If the fragment passes the depth test than it is processed by the fragment processor, else it is culled and not allowed to enter the fragment processor. In our case, in the depth buffer we provide 1.0 value to the depth component for non-zero projections, while 0.0 value to the zero projections. Rendering is invoked such that the depth component of each of the incoming fragments have 0.5 value, so ”less than” test is true only for the fragments (or projections) for which non-zero FP value is expected.

SIMDization of fragment program : Each fragment processing in our implementation, results in the computation of the FP value for one projection point (or ray). However, wherever evident we have exploited the SIMD capabilities of the GPUs to process one fragment.

4.2.2 Backward projection

- *Basic Approach* The main challenge in backward projection is to evaluate the numerator and denominator of the OSC backward projection equation as mentioned in []. Once the numerator and denominator are computed voxels can be updated by simply carrying out the division and using the value. The backward projection equation is symmetrically applied to all the voxels, so in order to explain the approach it is sufficient to discuss how the numerator

and denominator are evaluated for a single voxel. Since the forward projection values for all the projections are computed before carrying out the backward projection, its correct to update the voxels individually.

The evaluation of numerator and denominator of OSC backward projection equation ([1]) for a voxel requires the set of projection points or rays which effect this voxel. Apart from set of projections, we also need the intercept of those projections inside the voxel. Once such a set of projections along with its intercept is identified, the numerator and denominator can be evaluated. We identify such projections set for the voxel by drawing (one ray for each source-detector pair or projection angle) rays from the source through the center of the voxel under consideration. If the ray hit the exact grid point in the projection space, we take that projection in the set. If it doesn't hit an exact grid point, we use the interpolation (either bilinear or nearest neighbors) on the projection space. This kind of ray tracing operation gives us the set of projection points which are used to evaluate the numerator and denominator. The length of the rays inside the voxel are computed by using an optimized approach exploiting the fact that rays are passing through the center of the voxel. The Figure 4 explains our approach for length computation.

Algorithm 4.3 shows the pseudo code in the algorithmic terms along with the comments on the hardware resources utilized.

Algorithm 4.3 OSC Reconstruction : Backward Projection

```
OSCBP ( currVol, fpProj, expProj, subset )
begin
    PackData ( fpProj, expProj, subset ) ;
    Transfer all projections ∈ subset to GPU ;
    For ∀ zSlice ∈ currVol
        Set depth buffer ;
        Enable depth test ;
        For ∀ projection ∈ subset
            Set input values to shaders ;
            Render quadrilateral (Invoke computation for Numerator and denominator) ;
        End For
        Render quadrilateral (Invoke computation for division) ;
```

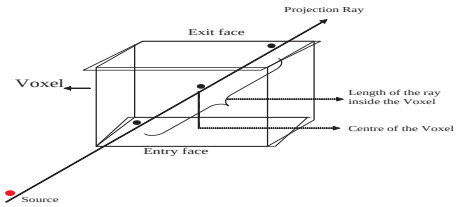

Figure 4: Length of the intercepted ray inside a voxel

Readback BP results for zSlice ;
End For
end

- *Optimizations* Several optimizations have been incorporated to use features available on GPU. This subsection discuss the details of the optimizations incorporated.

  Texture access optimizations by data packing : This optimization is a bit different from the data duplication optimization done in the context of FP. In this case since we need both FP value and the experimental value for all projections in a symmetric way, we pack them together so that a single access to the texture get us both FP and experimental values for the projection point. However, maximum number of texture access required can be four to carry out the bilinear interpolation.

Data transfer optimizations : The back projection is done slice by slice in the volume space using one subset of the projection. The entire subset of the projection is transfered once and than it is used to back project all the voxel slices.

Zero voxel initialization and Z-culling : This is an important optimization for BP. First of all, we initialize some of the voxels in the reconstructed volume to zero as shown in Figure 5. It essentially captures the fact that if any of the projection ray passing through a voxel has zero experimental value, than that voxel will have zero value in the final reconstructed volume. So, such voxels can be initialized to zero once and may not be considered in the reconstruction iterations. This selectively zero initialization of voxels is achieved by separate rendering passes on the GPU and done only once. All other voxels (which are not initialized to zero) will have non-zero initial value. We observe that voxels with zero value don't change their value throughout the OSC iterations and hence can be ignored from processing in a way similar to ignoring the zero value projection points in FP. This optimization is implemented using the same depth test technique (as used in FP) with non-zero voxels used to set the depth buffer.

Distributing computation across different stages of OpenGL pipeline : In our implementation we have distributed the BP computation across different stages of the pipeline. There are three main computation parts involved in the BP. Firstly, computing the projection space co-ordinate for a voxel. Secondly, using the projection values to evaluate the numerator and denominator of the OSC equation. Lastly, performing the division and updating the voxels.

The first part is equivalent to aligning the co-ordinate axes which are parallel to the volume faces to the co-ordinate axes which are parallel to detector sides, and than perspectively projecting the voxel onto the detector space. The vertex processor and the rasterizer is well suited to carry out this operation and we use them in our implementation. The second and third parts of the computation are carried out on the fragment processor of the OpenGL pipeline.

SIMDization of shaders : Each voxel is processed as a fragment in the fragment processor. Wherever evident, we utilized the RGBA SIMD capabilities of the GPU to process a fragment.

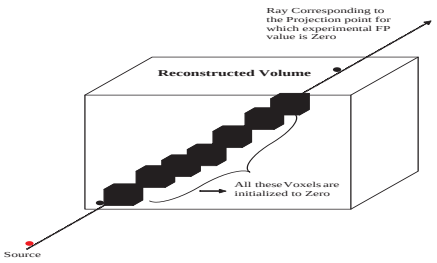

Figure 5: Zero voxel initialization

Algorithmic optimization for length inside the voxel computation : Length of intercept with the voxel for the projection ray inside the voxel is needed in the OSC equation. This can be computed by finding the intersection co-ordinate of a line with the voxel using co-ordinate geometry. However, this is computationally costly. We observe that since the rays are drawn from the source through the centre of the voxel, the entry and the exit point must lie on the parallel faces of the voxel. Further, since voxels are of unit dimension the maximum change in any co-ordinate between the entry and the exit point can be and will be unit. So, it follows that the faces of the voxel which are parallel to the axis along which the gradient of the ray is maximum will form the entry and exit faces. Once we have the entry and exit faces for a ray inside the voxel it is simple to find the intersection co-ordinates and subsequently the length.

5 Results and Discussions

The implementation for both Feldkamp and OSC algorithm were tested and benchmarked on Shepp-Logan phantom [8]. The GPU used for experimentation was Nvidia Quadro FX4600 [9]. The base line numbers were obtained by a single threaded run of the algorithms on Intel T2600 [10]. The size of reconstructed volume was 128 x 128 x 128, while the projection size was 128 x 128. We used 180 projections for running Feldkamp algorithm while 90 of them for OSC algorithm. We obtain 60 and 100 times speed-up for Feldkamp and OSC algorithms respectively on the chosen GPU using our implementation compared to the chosen baseline. It it to be mentioned here that the GPU used in the experimentation is not the most advanced GPUs available in the market. So, performance numbers are likely to improve on more powerful GPUs or by incorporating further algorithmic or architecture specific optimizations. Hence, in this report we have discussed our experience of implementing two of the important medical imaging algorithms on GPU.

References

[1] Fang Xu and Klaus Mueller *Real-time 3D computed tomographic reconstruction using commodity graphics hardware* Physics in Medicine and Biology 2007

[2] Kole, J.S. and Beekman, F.J. *Evaluation of accelerated iterative X-ray CT image reconstruction using floating point graphics hardware* IEEE Nuclear Science Symposium Conference Record, 2004

[3] Knaup, M. ; Kalender, W.A. and Kachelrieb, M. *Statistical Cone-Beam CT Image Reconstruction using the Cell Broadband Engine* IEEE Nuclear Science Symposium Conference Record, 2006

[4] H. P. Hiriyannaiah *X-ray Computed Tomography for Medical Imaging* IEEE Signal Processing Magazine, Vol. 114, No. 2, pp. 42-59, 1997

[5] L.A. Feldkamp ; L.C. Davis and J.W. Kress *Practical cone beam algorithm* J. Opt. Soc. Amer., Vol. A1, pp. 612619, 1984

[6] A.C. Kak and M. Slaney *Principle of computerized tomographic imaging* New York: IEEE Press, pp. 99-107, 1988

[7] M. Sakamoto ; H. Nishiyama ; H. Satoh ; S. Shimizu ; T. Sanuki ; K. Kamijoh and A. Watanabe *An implementation of Feldkamp algorithm for medical imaging on Cell* GSPx 2005 Multicore Application Workshop, Santa Clara, CA, 2005

[8] Jain, Anil K. *Fundamentals of Digital Image Processing* Englewood Cliffs, NJ, Prentice Hall, 1989, p. 439

[9] $http://www.nvidia.com/object/quadro_fx_5600_4600.html$

[10] $http://ark.intel.com/products/27237$

[11] $http://www.opengl.org/$

[12] $http://www.nvidia.com/object/cuda\_home\_new.html$