Performance improvements for iterative electron tomography reconstruction using graphics processing units (GPUs)

W.J. Palenstijn[a], K.J. Batenburg[a,b], J. Sijbers[a]

[a] *IBBT-Vision Lab, University of Antwerp*
Universiteitsplein 1, B-2610, Wilrijk, Belgium
[b] *Centrum Wiskunde en Informatica*
Science Park 123, NL-1098XG, Amsterdam, The Netherlands

Abstract

Iterative reconstruction algorithms are becoming increasingly important in electron tomography of biological samples. These algorithms, however, impose major computational demands. Parallelization must be employed to maintain acceptable running times. Graphics Processing Units (GPUs) have been demonstrated to be highly cost-effective for carrying out these computations with a high degree of parallelism. In a recent paper by Xu et al. [1], a GPU implementation strategy was presented that obtains a speedup of an order of magnitude over a previously proposed GPU-based electron tomography implementation. In this technical note, we demonstrate that by making alternative design decisions in the GPU implementation, an additional speedup can be obtained, again of an order of magnitude. By carefully considering memory access locality when dividing the workload among blocks of threads, the GPU's cache is used more efficiently, making more effective use of the available memory bandwidth.

Keywords: Electron Tomography, Reconstruction, GPU

Recently, iterative algebraic methods, such as ART and SIRT, have gained popularity in the electron tomography community due to their flexibility with respect to the geometric parameters of the tilt series, and their ability to handle noisy projection data. The use of algebraic reconstruction methods imposes major computational demands. Depending on the number of iterations, reconstructing a large 3D volume with a sequential implementation can easily take days on a normal PC. This obstacle can be largely overcome by parallelizing the computations, in particular the projection and backprojection steps. Graphics Processing Units (GPUs) have recently emerged as powerful parallel processors for general-purpose computations. Their architecture allows operations to be performed on a large number of data elements simultaneously.

Several algorithmic strategies have already been proposed for implementing algebraic methods for electron tomography on the GPU. In [2], it was demonstrated by Castaño-Diez et al. that at that time, a GPU implementation of the SIRT algorithm could achieve similar performance to a CPU implementation running on a medium sized cluster. Xu et al. recently proposed a different implementation strategy [1] that leads to a speedup of an order of magnitude compared to the results from [2]. They attribute this speedup to improvements in three categories: minimizing synchronization overhead, encouraging latency hiding, and exploiting RGBA channel parallelism. The first two design goals are interdependent and cannot be optimized separately. In this technical note, we argue that by exploiting data locality more effectively, the runtime of the projection and back projection operations can be substantially reduced, even though the required number of thread synchronization steps will increase. We demonstrate that a significant speedup can be gained in this manner.

Exploiting data locality

The Graphics Processing Unit (GPU) is well suited for carrying out the computations involved in electron tomography with a high degree of parallelism. A general outline of the processing and

memory-related capabilities of the GPU in the context of tomography is given in [1]. Here, we briefly discuss the features that are particularly relevant to our proposed implementation. Our implementation is based on the NVIDIA CUDA platform [3]. The GPU consists of a number of multiprocessors, each of which executes one, or a few, *thread blocks*. Threads on a multiprocessor share a texture cache. As the bandwidth available for reading data from the large global memory is limited, and tomography algorithms are highly memory-read intensive, it is imperative to exploit the possibilities for memory sharing and caching, even if this is at the cost of limited extra synchronization or extra memory writes.

To reduce the running time of iterative reconstruction algorithms, the projection and backprojection operations must both be accelerated. From this point on, we mainly focus on the projection operation. The general optimization strategy we propose also applies to backprojection.

In our implementation of the projection operation, each thread processes a single ray that is traced through the image to compute the corresponding detector value. This scheme ensures each memory location in the output projection image will be written to exactly once, so it cannot occur that the same output value is written to by two or more threads simultaneously. We use the *slice-interpolated* projection scheme, which yields similar accuracy to the grid-interpolated scheme employed in [1]; see [4]. Also, the computational efficiency of both models is comparable. The image is accessed through the texture unit and its corresponding cache memory. The performance of the shared texture cache depends on the 2D locality of the image values that must be loaded to compute the detector values for all lines processed within a thread block, as this cache is shared between all threads of a thread block (and between thread blocks on a multiprocessor). We use large thread blocks to fully exploit this locality, where each multiprocessor processes only a few such thread blocks.

In the article [1] by Xu et al., the rays are distributed among threads internally by the rendering driver software. By explicitly controlling the distribution of rays, or even segments of rays, among the thread blocks, the performance of the texture cache can be increased, thereby reducing the number of global memory reads required. We consider four ray ordering schemes. Starting from a fully unordered scheme, the efficiency of the texture cache is gradually improved:

(a) **No specified ordering** of the rays within thread blocks; see Fig. 1(a).

(b) Grouping **strips of parallel rays** within thread blocks. For each projection angle, rays corresponding to neighboring detectors follow neighboring, parallel paths through the volume. If their threads are part of the same thread block, they will execute in parallel, and will access nearby locations in the image texture; see Fig. 1(b).

(c) Grouping strips of rays for several **consecutive angles** within thread blocks. For sets of consecutive projection angles, rays corresponding to a fixed set of neighboring detectors will partially overlap within the slice; see Fig. 1(c).

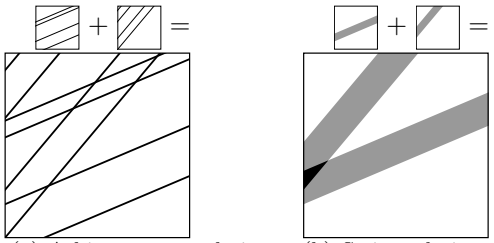
(d) Grouping **strip segments** for several angles within thread blocks. As the angle between two projections increases, the overlap that can be exploited between strips is reduced. To exploit this overlap more effectively, the strips of rays are subdivided into *strip segments*, as shown in Fig. 1(d). Based on spatial locality, sets of strip segments (spanning across several projection angles) that have substantial overlap are processed within a single thread block. The complete detector values can then be obtained by accumulating the partial values from the segments.

In our implementation, we use a voxel driven backprojector: for each voxel in the 3D volume, the total backprojected value is computed by accumulating the corresponding sinogram values for all angles. Each GPU thread processes a single voxel. Similar considerations as above apply to the grouping of voxels within thread blocks, as grouping local regions of voxels within a slice also results in locality in the sinogram domain. While all three steps above improve locality for the backprojector, selecting appropriate segments in scheme (d) has a rather high computational cost outweighing the benefits. Our backprojector therefore implements scheme (c).

Experiments and results

To investigate the impact of each of the three ray ordering improvements on the running time of the projection operation, we performed a series of experiments.

Figure 2: Subset of the sinogram processed by a single thread block for the different scenarios.



(a) Arbitrary ray ordering

(b) Strip ordering; arbitrary angles

(c) Strip ordering; consecutive angles; high overlap

(d) Strip segments; increased overlap

Figure 1: Different scenarios for selecting the set of rays processed within a single thread block. The square region represents a volume slice orthogonal to the tilt axis. The small squares depict the subsets of these rays corresponding to two different projection angles. The large squares show the resulting spatial overlap.

Test case (a) is shown in Fig. 2(a). The background of this figure (and the subsequent ones) shows a *sinogram* of a slice of the 3D volume, where the horizontal axis corresponds to the coordinate at the detector and the vertical axis corresponds to the projection angle. The black overlay indicates the rays that are combined within a single thread block. In test case (a), a GPU thread block handles 16 angles, and for every angle it processes 32 rays that are distributed at constant intervals covering the entire detector. The angles are similarly chosen at constant intervals, covering the complete angular range. This results in a ray distribution in the image domain that has similar data locality properties as Fig. 1(a).

Test case (b) is shown in Fig. 2(b). The distribution of angles over the thread blocks is the same as in the previous case, while each thread block now handles a strip of 32 consecutive rays, resulting in a pattern similar to Fig. 1(b).

In test case (c), we group 16 consecutive angles and 32 consecutive rays in each thread block; see Fig. 2(c) and Fig. 1(c).

Test case (d) is significantly more complex than the previous ones. Each ray is subdivided into 8 segments. Strip segments are formed by grouping 32 neighboring ray segments for a particular angle. A thread block processes a group of strip segments for 16 consecutive angles, selecting strip segments that have large overlap. Fig. 1(d) depicts the increased locality in this scheme. The thread blocks for all segments that constitute a ray are processed sequentially — accumulating the partial detector values computed — thereby ensuring that multiple threads never write to the same memory location simultaneously.

We will now report the results of a series of experiments, carried out to investigate the effect of the different ray ordering schemes. The results for the fastest ordering scheme will subsequently be compared to the results reported in Xu et al. [1].

Two generations of NVIDIA GPUs have been used. The GTX 280 GeForce card contains a GT200 GPU and was launched in 2008. The same card was used by Xu et al. The GeForce GTX 480 and Tesla C2070 cards contain a Fermi GPU, and were launched in 2010.

For the reconstruction and timings we did not use the long object compensation method also described in [1]. Since it is essentially a preprocessing step, it does not affect the timings of the iterations.

For the experiment comparing different ray ordering schemes, we have run the projection operation on a stack of 10 slices of size 2048×2048, using 180 projection angles at 1° intervals. Table 1 contains the projection time per slice for the test cases 0–3 described above.

We now report running times for the SIRT and OS-SIRT 5 algorithms; see Xu et al. [1] for details on these algorithms. Both implementations use the

Test case	GTX 280 (ms)	GTX 480 (ms)
(a)	507.103	278.622
(b)	154.976	85.068
(c)	77.863	34.340
(d)	25.326	27.883

Table 1: Projection timings per slice for the four test cases: (a) Non-consecutive rays and angles; (b) Strips of rays at non-consecutive angles; (c) Strips of rays at consecutive angles; (d) Strip segments with high overlap.

Slice size/angles	SIRT GTX 280 (ms)	OS-SIRT 5 GTX 280 (ms)	SIRT GTX 480 (ms)	OS-SIRT 5 GTX 480 (ms)
512^2, 180 angles	3.8	9.7	3.5	7.6
512^2, 360 angles	6.8	11.9	6.4	10.2
1024^2, 180 angles	12.4	30.0	12.9	22.4
1024^2, 360 angles	22.9	36.0	23.8	34.2
2048^2, 180 angles	45.1	110.9	48.0	73.7
2048^2, 360 angles	84.8	124.7	91.0	118.5
4096^2, 180 angles	171.5	416.9	184.1	263.5
4096^2, 360 angles	326.5	459.6	354.0	438.9

Table 2: Timings per slice for one iteration of SIRT and 5 iterations of OS-SIRT 5.

proposed projection and backprojection operations: the projection employs scheme (d), while the backprojection employs the analogue of scheme (c). We have run the algorithms on stacks of 10 slices of different sizes, using 180 projection angles at interval 1° and also 360 projection angles at interval 0.5°. The running time per iteration was computed by running the algorithm for 10 and 50 iterations and taking 1/40th of the difference in execution time. The results of these experiments are shown in Table 2.

In Table 3, SIRT timing results for our proposed implementation are compared with the results reported in Table 1 and Table 2 of [1], using matching slice dimensions.

The results in Table 1 clearly demonstrate the benefits of increasing the locality of the memory access scheme, with each of the optimization steps providing a significant speedup.

Table 2 and the corresponding Fig. 3 provide insight in how our implementation scales to larger image sizes. It can be observed that the amount of time spent for each voxel increases slightly sublinearly with the size of the reconstructed volume. For OS-SIRT 5, processing the same number of projections as for SIRT results in a longer running time, which can be explained by the larger average gaps between the projection angles, reducing data locality. This effect is smaller when using 360 projections than for the case of 180 projections, which supports this explanation.

Slice size	Xu et al. GTX 280 (ms)	This paper GTX 280 (ms)	GTX 480 (ms)	C2070 (ms)
356 × 148	4.2	0.9	0.7	1.2
712 × 296	10.5	2.0	1.8	2.7
1424 × 591	28.8	5.5	5.6	7.6
512 × 512	34.8	3.8	3.5	5.1
1024 × 1024	130.3	12.4	12.9	17.3
2048 × 2048	547.0	45.1	48.0	63.6

Table 3: Timings per slice for one iteration of SIRT. The three non-square slices are based on 61 projections and a tilt range of 120 degrees. The square slices are based on 180 projections and a tilt range of 180 degrees.

Compared to the results reported in Xu et al. [1], our implementation of SIRT is about ten times as fast when run on a GTX 280 graphics card; see Table 3. This time difference is within the range of timings observed when comparing the different ray ordering schemes in Table 1. Other factors, such as the particular GPU programming platform used and low-level coding details will also play a role in the observed timing difference, such that it is not possible to attribute the results solely to the ray ordering scheme. Still, our results demonstrate clearly that exploiting data locality is a crucial factor in the total running time.

In the three tables, various remarkable differences between the performance of the GT200 (GTX 280) and Fermi (GTX 480 and Tesla C2070) GPUs can be observed. Although the Fermi architecture has a higher degree of parallelism, and is therefore computationally faster, the balance between the speeds of different components (e.g., texture cache, arithmetic units, memory bus) has also changed substantially. As a consequence, the Fermi cards are actually slower than the older GTX 280 for some particular cases. This is particularly noticeable for the SIRT algorithm.

Finally, we illustrate that our implementation of SIRT produces proper reconstructions by means of Fig. 4. It shows part of a reconstructed slice of a sample of keyhole limpet hemocyanin (KLH) in vitrified ice. The data was recorded on an FEI Titan Krios with 6–10 μm defocus, zero-loss filtered with a 20 eV slit, in low-dose mode, with 71 projections and a tilt range of 140°.

In conclusion: We have demonstrated that the ray segment ordering scheme has a major impact on the running time of the projection and backprojection operations in a GPU implementation. For our implementation, a speedup of around 10 was observed compared to recent results reported by Xu et al. using similar hardware.

Acknowledgments: The authors wish to thank Jeff Langyell from FEI Company for recording and providing the KLH dataset. We gratefully acknowledge IBBT, Flanders for financial support.

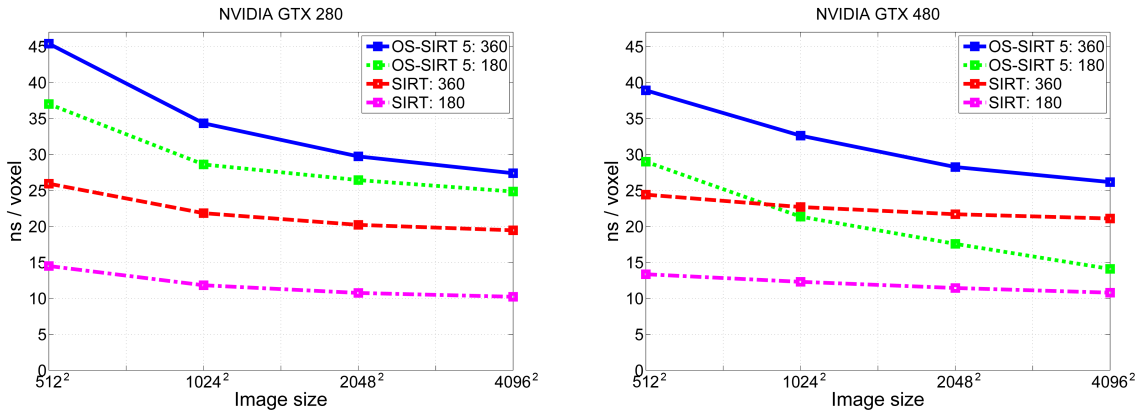


Figure 3: Comparison of the time spent per voxel for the SIRT and OS-SIRT 5 algorithms, for the GTX280 and GTX480 GPU architectures.

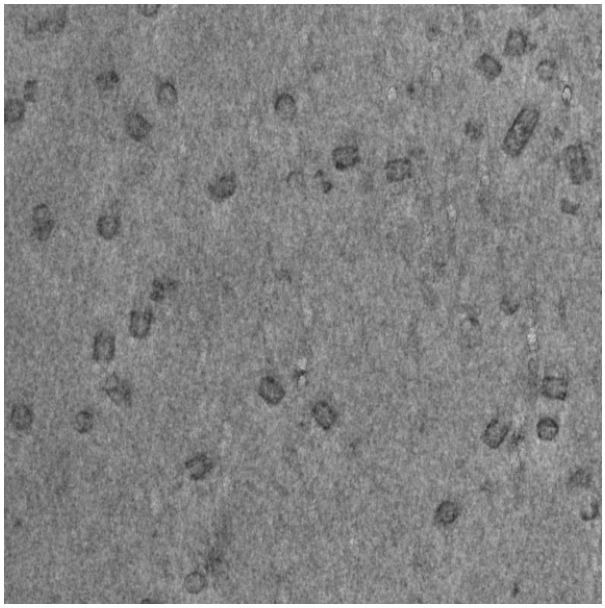


Figure 4: Part of a reconstructed slice from the KLH dataset

References

[1] W. Xu, F. Xu, M. Jones, B. Keszthelyi, J. Sedat, et al., High-performance iterative electron tomography reconstruction with long-object compensation using graphics processing units (GPUs), Journal of Structural Biology 171 (1) (2010) 142–153.

[2] D. Castaño-Diez, H. Mueller, A. S. Frangakis, Implementation and performance evaluation of reconstruction algorithms on graphics processors, Journal of Structural Biology 157 (1) (2007) 288–295.

[3] NVIDIA CUDA C Programming Guide, Version 3.2 (November 2010).

[4] F. Xu, K. Mueller, A comparative study of popular interpolation and integration methods for use in computed tomography, in: Proceedings of the IEEE 2006 International Symposium on Biomedical Imaging (ISBI 2006), 2006, pp. 1252–1255.