ASSEMBLING LARGE MOSAICS OF ELECTRON MICROSCOPE IMAGES USING GPU

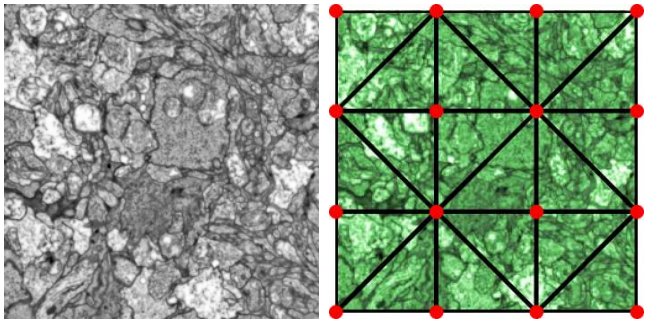
Kannan Umadevi Venkataraju*†, Mark Kim*†, Dan Gerszewski†, James R. Anderson‡ and Mary Hall†
*Scientific Computing and Imaging Institute, University of Utah
†School of Computing, University of Utah
‡Moran Eye Center, University of Utah

***Abstract*—Understanding the neural circuitry of the retina requires us to map the connectivity of individual neurons in large neuronal tissue sections and analyze signal communication across processes from the electron microscopy images. One of the major bottlenecks in the critical path is the image mosaicing process where 2D slices are assembled from scanned microscopy image tiles. The problem of assembling the tiles is computationally non-trivial because of distortion of the specimen in the electron microscope due to heat and overlap between the scanned tiles. The complexity of the calculation arises from the massive size of the dataset and mathematical calculations required to calculate value of each pixel of the mosaic. We propose to use texture memory lookups to speedup the access to image tiles and data parallel computing enabled by the GPUs to accelerate this process. The proposed method results in noticeable improvements in speed of computation compared to other methods.**

***Index Terms*—Serial-section TEM, image mosaicing, GPGPU, CUDA, texture**

I. Introduction

Novel imaging techniques are being used to map the connectivity of individual neurons in large neuronal tissue sections, to understand the neural circuitry of the retina, and particularly how signals are communicated across processes. Extensive studies have been undertaken using electron microscopy to create detailed diagrams of general neuronal structures [1] and their connectivities [2], [3], [4]. The entire volume of neuronal tissue is scanned as ultra thin slices (approx. 90nm) sliced using a micro tome and assembled to reconstruct the 3D volume. The neuronal tissue has to be scanned at nanoscale resolutions to unambiguously identify the neurons and synapses and create detailed maps. The serial-section Transmission Electron Microscope (TEM) is the preferred imaging modality for capturing large sections (0.25 mm^2) of neuronal tissues with synaptic and gap junction at high resolutions (2.18nm/pixel). Rarely do we find an electron microscope that can capture at the required nanoscale resolution and such a wide field of view. Thus the sample of interest is imaged as a sequence of tiles (Figure 1) with some overlap. The imaging of these tiles using TEM requires the sample to be suspended over a high energy beam of electrons causing it to heat up and subsequently distort. Since distortion is not uniform among tiles they have to be unwarped individually. Thus, reconstructing the image from the set of tiles, called image mosaicing, involves significant computation to identify and handle tile overlap and correct nonuniform distortions. A typical neuronal section is 2500 microns in diameter and scanned as 1000 tiles. Currently, researchers assemble the volume from the scanned tiles using a multi-threaded tool chain [5], but this computation is one of the bottlenecks in the critical path to reconstruct the volume since it is estimated to take around 90 days to assemble full mosaic images for 340 mosaics.



(a) (b)

Fig. 1. Example serial-section TEM images: (a) Sample electron microscope scanned tile of neural tissue of mice and (b) Tile with triangle mesh overlay

Image mosaicing is the process of stitching multiple images into a single mosaic image such that the corresponding points of the individual images match. In our problem, we use deformable transforms [7] to model the nonuniform distortions in the image tiles. Bui et al. [6] have shown that GPUs give good speed up on similair image registeration tasks.

The method proposed in this paper improves upon previous work by utilizing GPU as computational unit. In this paper, we describe our experiences using GPUs to accelerate this computation. Because of the inherent parallelism of the computation, the roughly identical computation at each pixel, and the data locality across neighboring tiles, our initial observation was that this computation ought to achieve high speedup if we can effectively harness the arithmetic and bandwidth capabilities of the GPU. The remaining sections of the paper gives an overview entire computation, details of the implementation finally followed by a discussion on results.

II. Methods

In this section we introduce the mosaicing algorithm. The deformable transformation undergone by the tile is modeled as an uniform grid transform. The control points of the image tile are calculated by sampling the image by an uniform grid. The location of the control points' correspondences on the final mosaic are determined by an image registration process and stored in a transform file [7]. Our implementation reads in the transform file and creates a triangle mesh out of the control points for every tile in the mosaic as shown in Figure 1.

The application then calculates the span of the mosaic and calculates the pixel Grey value for every pixel of the mosaic image as described below.

1) The image tiles contributing to pixel are determined.
2) Every triangle (from all contributing tiles) in the mesh contributing to the point is determined.
3) Using barycentric co-ordinate system the location of the point on the deformed tile is calculated.
4) If there are multiple image tiles contributing to the pixel then the final pixel value is calculated by a convex blend of these values. This is known as feathering

Thus we see that the computation intrinsically has high data level parallelism. NVIDIA's CUDA has the ability to perform such non-graphics data-parallel computation on the GPU easier without the need of mapping the computation to graphics APIs. The following section describes the CUDA implementation of the application.

III. Implementation

In this section firstly, we discuss the computations implemented as CUDA kernels. Then we discuss the major optimizations used to achieve the speedup.

A. Kernels

The data-parallel portions of the algorithm were identified and implemented as CUDA kernels.

1) Triangle search: This kernel does a linear search on all the deformed triangles to determine if the point of interest lies within the the triangle. This is verified by determining if the point lies within the convex hull formed by the three vertices as described below.

$$a = \frac{det(vv_2) - det(v_0v_2)}{det(v_1v_2)}, b = \frac{det(vv_1) - det(v_0v_1)}{det(v_1v_2)} \quad (1)$$

where,

$$det(uv) = u \times v = u_xv_y - u_yv_x \quad (2)$$

and v_0 is a vertex, v_1, v_2 are vectors from v_0 to other two vertices and v is the point of intrest.

If a,b $>$ 0 and a+b $<$ 1, then the point lies inside the triangle [8].

2) Projection: Once the triangle containing the point P is determined, the point is projected to the undeformed triangle by a barycentric coordinate system as follows. Let the undeformed traingle be $\triangle\ ABC$ and P be the point inside. Let the corresponding deformed traingle be $\triangle\ A'B'C'$ and P' be the point inside. The projected point P' is given by the following equation

$$P' = \lambda_1A' + \lambda_2B' + \lambda_3C' \quad (3)$$

where,

$$\lambda_1 = \frac{Area(\triangle PBC)}{Area(\triangle ABC)}, \lambda_2 = \frac{Area(\triangle APC)}{Area(\triangle ABC)} \quad (4)$$

$$\lambda_3 = \frac{Area((\triangle ABP)}{Area((\triangle ABC)} \quad (5)$$

3) Interpolation: Once the point location is calculated, the Grey value of the pixel at that point is determined by nearest neighbor interpolation. The texture memory to used to speed up this interpolation.

B. Optimizations

In this section we first discuss how the interpolation is accelerated, followed by mechanism used to streamline the image data transfer between CPU and GPU.

1) Texture optimization: One of the most effective schemes to get the most performance out of CUDA is coalesced memory reads from global memory to shared memory. Once data is in shared memory, accessing and manipulating the data in shared memory has the same access time as a register [9]. In the case of our mosaicing algorithm, coalescing memory reads proves to be difficult due to the nature of the the algorithm itself. Coalesced data reads requires that the data fetched from global memory be accessed in a linear fashion on particular memory boundaries. However, when we fetch multiple pixel values from an image tile, there is no guarantee that the pixels accessed by the warp will be in a linear fashion. In fact, although the pixels fetched would have a strong locality, because of the transformation the locality is in two dimensions, which ruins the access pattern for global memory.

Instead of using global memory, we used texture memory. Texture memory is very similar to global memory with one crucial difference: texture memory has an 8KB cache separate from registers or shared memory. Also, the caching scheme is optimized for 2D spatial locality [9]. As stated before, the access pattern for our algorithm has strong spatial locality in two dimensions and the texture cache handles this automatically. Further, because we are not changing the data within a tile, texture memory is a good choice as well. The texture memory paradigm does not allow for writing to texture memory and then accessing it again within the same kernel call. Since we do not do this, texture memory is a good choice. Thus, texture memory is strongly suited for our particular algorithm.

2) Texture queue: The use of a queue has a two fold purpose. First, the most memory available on a single processing card is currently 4GB [10]. Obviously on a fullsized mosaic, 4GB would be woefully insufficient. The queue enables us to only put into texture memory the image tiles that are currently worked on and not worry about squeezing all 1000 image tiles onto the card. Further, because of the overlap, multiple image tiles may be required for calculating value of one pixel. The queue allows us to consider the transformed triangles and place into texture memory the correct image tiles that are needed by the currently processed triangle. Depending on the amount of memory available on the processing card, this also allowed us to create a multi-pass mosaicing system: if available memory is less than the total amount required for the number of image tiles needed for the triangle processed, then we process however many image tiles fit into memory and then run the algorithm again and load up the remaining image tiles to be processed. This allows us to be flexible with the types of cards that can be used.

IV. Results

The mosaicing application was benchmarked against other applications using a mice neural tissue dataset. The final mosaic is 13783×13686 pixels in size. It is made up of 16 tiles of size 4080x4080 pixels each. The control point grid is 8×8 in size. The mosaic image along with few deformed tile images is shown in Figure 2. The CUDA application was benchmarked against a single threaded equivalent, multi threaded application (using OpenMP) and finally against a highly optimized ITK based multithreaded application [5]. It was run on a Intel Core 2 Quad CPU Q9550 @ 2.83 GHz PC running 64-bit SuSE linux. The CUDA application was run on the same PC on its NVIDIA GeForce GTX 280 graphics card. From the benchmarking results are shown in Table I, we can infer that the GPU based acceleration is highly suitable this specific image mosaicing operation and for complex image processing applications in general.

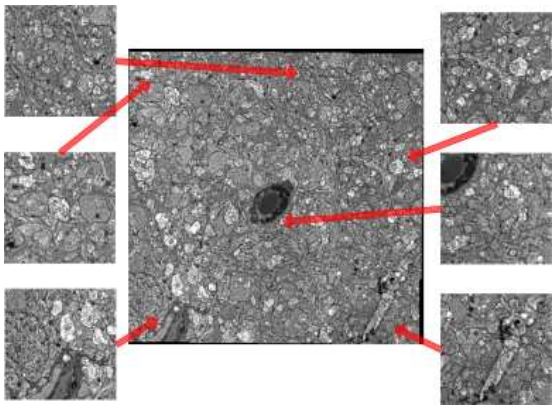


Fig. 2. Mosaic image and few tiles

V. Conclusion

The proposed method utilized the GPU architecture to speed up the image mosaicing process. Along with the texture lookup and streamlined data transfer between GPU and CPU, this program provides better acceleration. Thus one can expect similair speed up on larger datasets. However, the usage of unsigned char as the image data type, the method is limited to use nearest neighbor interpolation. This may result in slightly inaccurate calculation of the pixel values in the mosaic. This problem could be avoided by using float data types and other smoother interpolation techniques like bilinear, bicubic or spline methods. Future work would address these problems in pixel value calculation.

TABLE I
Bench marking results

Programming model	**Time elapsed(in seconds)**	**Speed Up**
Single threaded C	2022.3	N/A
OpenMP multi threaded (16 threads)	1140.46	1.77x
ITK based multi threaded [5]	120	16x
NVIDIA CUDA	10.8	187.23x

Acknowledgment

The authors would like to thank the members of the Collaborative Research in Computational Neuroscience (CRCNS) group at the Scientific Computing and Imaging Institute and the Marc Lab for the useful comments on the project. We also like to thank NIH R01 EB005832 (PI Tasdizen) grant, DOE VACET, KAUST GPR: KUS-C1-016-04, NSF: CNS-0615194, CNS-0551724, CCF-0541113, and IIS-0513212 for enabling us to do this work.

References

[1] J. C. Fiala and K. M. Harris, "Extending unbiased stereology of brain ultrastructure to three-dimensional volumes," *J. Am. Med. Inform. Assoc.*, vol. 8, no. 1, pp. 1–16, 2001.

[2] J.G. White, E. Southgate, J.N. Thomson, and F.R.S Brenner, "The structure of the nervous system of the nematode caenorhabditis elegans," *Phil. Trans. Roy. Soc. London Ser. B Biol. Sci.*, vol. 314, pp. 1–340, 1986.

[3] Kevin L. Briggman and Winfried Denk, "Towards neural circuit reconstruction with volume electron microscopy techniques," *Curr. Opin. Neurobiol.*, vol. 16, no. 5, pp. 562–570, Oct. 2006.

[4] J.R. Anderson, B.W. Jones, J.-H. Yang, M.V. Shaw, C.B. Watt, P. Koshevoy, J. Spaltenstein, E. Jurrus, Kannan U.V., R.T. Whitaker, D. Mastronarde, T. Tasdizen, and R.E. Marc, "A computational framework for ultrastructural mapping of neural circuitry," *PLoS Biology*, vol. 7, no. 3, pp. e74, 2009.

[5] P. Koshevoy, T. Tasdizen, R.T. Whitaker, B. Jones, and R. Marc, "Assembly of large three-dimensional volumes from serial-section transmission electron microscopy," in *Proceedings of 2006 MICCAI Workshop on Microscopic Image Analysis with Applications in Biology*, October 2006.

[6] Peter Bui and Jay B. Brockman, "Performance analysis of accelerated image registration using gpgpu," in *GPGPU*, 2009, pp. 38–45.

[7] L. Ibanez, W. Schroeder, L. Ng, and J. Cates, *The ITK Software Guide*, Kitware, Inc. ISBN 1-930934-15-7, http://www.itk.org/ItkSoftwareGuide.pdf, second edition, 2005.

[8] "Triangle interior," http://mathworld.wolfram.com/TriangleInterior.html.

[9] "Nvidia cuda compute unified device architecture programming guide," http://developer.download.nvidia.com/compute/cuda/2_2/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.2.pdf.

[10] "Nvidia tesla c1060 computing processor," http://www.nvidia.com/object/product_tesla_c1060_us.html.