SAND2012-9209C

An Irregular Approach to Large-Scale Computed Tomography on Multiple Graphics Processors Improves Voxel Processing Throughput

Edward S. Jimenez, Laurel J. Orr, and Kyle R. Thompson

Sandia National Laboratories
PO Box 5800
Albuquerque,NM 87185
{*esjimen,ljorr,krthomp*}*@sandia.gov*

***Abstract*—While much work has been done on applying GPU technology to computed tomography (CT) reconstruction algorithms, many of these implementations focus on smaller datasets that are better suited for medical applications. This paper proposes an irregular approach to the algorithm design which utilizes the GPU hardware's unique cache structure and employs small x-ray image data prefetches on the host to upload to the GPUs while the devices are operating on large contiguous subvolumes of the reconstruction.**

This approach will improve the overall cache hit-rates and thus improve the performance of the massively multi-threaded environment of the GPU. Overall, utilizing small prefetches of x-ray image data improved the volumetric pixel (voxel) processing rate when compared to utilizing large data prefetches which would minimize data transfers and kernel launches. Additionally, this approach does not sacrifice performance on small datasets and is thus suitable for medical and industrial applications. This work utilizes the CUDA programming environment and Nvidia's Tesla GPUs.

I. Introduction

Cone-beam Computed Tomography (CT) is an indirect 3D imaging technique in which a set of 2D x-ray projection images are used to reconstruct the internal and external structure of the imaged object [1]. Many industrial applications of Cone-beam Computed Tomography aquire a very large number (usaually greater than 900) of x-ray projection images taken around an axis of rotation [2]. Large-sized reconstructions for this work can mean the x-ray image projections are numerous, the x-ray projections are high-resolution, the volume to be reconstructed is high-resolution, or any combination thereof.

CT reconstructions require significant computation and, in many cases, expensive computing resources. The popular FDK (Feldkamp) 3D reconstruction algorithm has computational complexity $O(n^4)$ [3], [4]; work by Xiao et. al. [5] and Axelsson et. al [6] have reduced the complexity to $O(n^3 \log(n))$, but even with this improvement, computation for large-scale datasets can still require days to weeks to complete using high-end workstations.

Over the past thirty years, various methods have been investigated to improve the computational speed of CT reconstructions including the use of supercomputers, dedicated hardware (ASIC, FPGA, etc), and commodity graphics processing units (GPU) [7]. Currently, the GPU is the most cost effective and versatile. Computed Tomography can be thought of simply as a set of pixel and voxel (volumetric pixel) operations and thus is not a far reach to consider the massively parallel architecture of the GPU. The GPU employs a SIMD (single instruction multiple data) programming model with large- and small-grain parallelism [8]. Driven mainly by the Gaming and CAD industry, investment in GPU-technology is unparalleled and has exhibited performance growth that triples Moore's Law [7].

There has been extensive work done on applying GPU technology to various CT algorithms [4], [7], [9], [10] with tremendous computation time improvements realized when compared to traditional CPU-based implementations. However, much of the development of the GPU algorithms focus on medical datasets which are typically an order of magnitude smaller than industrial CT datasets. The majority of medical CT datasets consist of $128^3 - 1024^3$ voxels and 300-1000 x-ray image projections. The number of acquired x-ray projection images and the resolution is balanced by the need to minimize radiation exposure to the patient.

For industrial CT applications, radiation exposure is typically not a concern as it is with medical applications and often object density will require long exposure times. With the exposure constraint relaxed, it is not uncommon

for industrial datasets to reach 4000^3 voxels and approximately 2000 projections. Unfortunately, much of the medical literature on GPU-based CT assume that either the volume, x-ray projection data, or both can entirely reside on the GPU device memory simulataneously. This is not possible in most large-scale (i.e. industrial) reconstructions and thus a blocking algorithm that blocks both the volume and the x-ray projection data is necessary.

This paper will present a technique for large-scale CT reconstruction that implements an irregular approach to the pixel and voxel operations that maximizes voxel throughput for large-scale datasets. Two large synthetic datasets will be presented; the first consists of 1800 16-megapixel x-ray projection images reconstructed into a $4000 \times 4000 \times 4000$ voxel volume (64 Gigavoxels); this case is representative of real-world industrial CT datasets, the second consists of 10,000 100-megapixel images reconstructed into a $10000 \times 10000 \times 10000$ voxel volume (1 Teravoxel); this dataset was chosen to show that this approach is capable of handling future-sized datasets.

II. Approach

When approaching a massively parallel problem, one must be aware of the various bottlenecks that are not typically present in single-threaded algorithms. One major bottleneck in GPU computing is the data transfer between host and device. The typical approach to allieviate this bottleneck is to minimize the total number of data transfers [11].

If one were to follow this scheme for the reconstruction of a given subvolume, then it would be desirable to fit large amounts of x-ray data per kernel launch and thus minimizing the number of x-ray data uploads to the GPU necessary to reconstruct the given subvolume. Furthermore, one could utilize host pinned-memory to maximize data transfer speeds [8]. On the surface, this should guarantee minimal interruption during voxel processing.

The issue with the approach described above is twofold. First, allocating large amounts of pinned-memory on the host is normally not allowed by the operating system. Second, for a given subvolume and a relatively large amount of x-ray data, the memory access pattern on the x-ray data may become scattered and thus hindering kernel performance.

Much of the work done in the past has addressed this by utilzing read-only texture memory which utilizes texture cache and fast bilinear interpolation [9]. Utilizing this approach for large-scale reconstruction still results in scattered reads and poor performance as the texture cache-hit rate is very low. Scattered memory access patterns are mostly caused by two factors. First, if the geometrical configuration of the imaging system is set up for significant magnification, then the interpolation coordinates for neighborhoods of voxels could be spread out over a large portion of a given x-ray image. Second, thread execution order could have a measurable effect on computationaly efficiency as different threads in a warp could potentially be accessing x-ray data from different projection images.

A. Irregular Approach

The combination of large-scale data, blocked x-ray data, and blocked subvolumes suddenly makes CT an irregular problem. CT algorithms transfer a large number of bytes from both the volume and the x-ray data, but are also very computationally expensive at $O(n^4)$. The massively parallel environment and imaging system configuration has the potential to create little data locality. Additionally, the amount of x-ray data necessary to reconstruct a given subvolume is dependent on the location of the subvolume with respect to the entire volume, and thus a dynamic approach to subvolume size determination is necessary. Traditional CT algorithms typically reconstruct by *slices*, which are defined as coplanar sets of voxels. In this work, a subvolume is a set of consecutively ordered slices and will be referred to as a *slice block*.

The approach presented does not focus on data transfer minimization, but instead, texture cache-hit rate improvement by reducing the amount of x-ray image data uploaded at once combined with data prefetching. A paper by Mowry and Gupta which looked at an irregular application showed that performance could be improved with an intelligent data prefetching approach which focused on improving the cache-hit rate of the application [12]. Additionally, work done by Lam et. al. showed that cache interference in blocked algorithms can have a significant performance degradation for a given machine [13]. Overall, five aspects of the algorithm design, which uses the CUDA programming environment, will be addressed.

1) Massive Parallelism: The computational intensity of the CT algorithm necessitates a massively parallel environment. For this application, a slice block with s slices and N voxels per slice will require N computational threads, where each thread is responsible for processing s voxels in the subvolume. More specifically, a thread is responsible for a column of voxels in the subvolume, one on each slice. The thread will loop over all images in the image subset present in the GPU memory for given slice before advancing to the next slice. This approach helps to keep the memory access pattern somewhat coalesced, potentially increasing the cache hit-rate, and also allows for only one voxel update to global memory per kernel launch.

2) Texture memory/Texture cache: This approach will utilize the Texture/L2/Global memory hierarchy available on the GPU to improve the bi-linear interpolations on the x-ray image as this is the main computational burden in the FDK algorithm. Utilizing texture memory for x-ray image data is not a new idea and is key to many GPU-based CT algorithms [4], [7], [9], [10]. However, this approach utilizes small texture memory allocations for the x-ray data in relation to the allocations used for the subvolumes so that a larger fraction of the texture memory fits within the texture and L2 caches. As fetches from texture and L2 cache are up to two orders of magnitude faster than fetches from GPU global memory, this approach will improve overall voxel processing throughput by decreasing the time to fetch information from the x-ray projection data as well as reducing memory traffic on the GPU global memory bus. Texture memory also has the benefit of allowing one to utilize fast hardware-based low-precision bilinear interpolation to improve computational speed.

3) Constant Memory: Constant memory on the GPU is another type of cache specific to GPU hardware that is user-specified. This cache is also orders of magnitude faster than global memory and is ideal for variables that are shared across threads. For this implementation, geometrical information about the imaging system that is needed for the reconstruction computation is stored here, further reducing the demand on the global memory bus.

4) Data Prefetching to Pinned-Memory: While the GPU device is operating on an x-ray image subset, the CPU is prefetching the next image subset to a pinned-memory region that will be uploaded to the GPU. The x-ray image dataset will already be loaded in main memory with the pinned-memory region being separate from the global x-ray data. Smaller pinned-memory allocations greatly increases the chance that the allocation will be successful. As mentioned earlier, pinned-memory increases data throughput on data transfers, and the prefetching while the kernel is executing will guarantee that pauses between kernel launches are kept to a minimum.

5) Dynamic Task Partioning: One desirable feature of this algorthim is for it to be scalable with respect to the number of GPUs present on the system. In order for this algorithm to be scalable from one to many GPUs, it must maximize all GPU memory resources to ensure that the GPUs are as busy as possible. It was mentioned above that the amount of x-ray data varies with respect to the location of the slices in the global reconstruction. Additionally, GPU memory values vary greatly between GPU models and configurations. This results in the need for a dynamic partitioning scheme. The overall partitioning approach will maximize the number of contiguous slices that can reside on a particular GPU and use the remaining memory available for the x-ray image data. This will determine the number of kernel launches necessary to fully reconstruct the subvolume on the GPU. If at least one x-ray projection image does not fit on the remaining memory, then the number of slices on the GPU is reduced by one and the process is retried. The minimum requirement for this algorithm is that the GPU fit at least one slice and its x-ray subimage that contains the partial projection image that is necessary for the computation.

6) Computation Ordering: When developing a kernel algorithm, one needs to be aware that accessing a register consumes zero extra clock cycles per instruction, but latencies may occur due to register read-after-write dependencies. At approximately 24 clock cycles for Nvidia GPUs, these latencies could be very significant when processing millions of voxels simultaneously [14]. The massive number of threads helps to cover this latency but may not be enough for all configurations. The instruction ordering of the kernel is designed such that it minimizes to the need to immediately access a variable it just computed as well as reducing the Register pressure to ensure that no values in regiser are being cached to the GPU global memory.

III. Implementation

This GPU-based approach is implemented using Nvidia's CUDA programming environment and C++. The kernels developed for this application are written such that any Nvidia graphics processor with at least 1 GB of device memory and at least Fermi architecture is capable of performing a reconstruction provided at least one slice and one x-ray image subset (consisting of at least one x-ray subimage) can reside in memory.

Other kernels developed, but not presented in this work, include slightly less efficient implementations that guarantee a kernel runtime of less than two seconds to allow GPUs that are subject to display timeout restrictions to run larger reconstructions. This implementation can allow for 1 to 8 GPUs to run on a single system using OpenMP 2.0. For this work, assume that all x-ray image data is resident on the host memory (this work makes no claims on disk I/O performance and will be addressed in future work).

The dynamic task partitioning is determined by a *slice-to-texture* ratio (STR) that is configured using a parameter in an input file. This ratio tells the application to attempt to fit the data on the device memory in such a way that the number of simulataneously reconstructed slices to the total number of image subsets satisfies the given ratio as closely as possible. There are three possible reasons why this ratio may not be satisfied exactly:

- **Resource Maximization:** The partitioning function will maximize device memory usage. Any ramaining memory after allocation will be utilized for additionaly x-ray image data. This was implemented since some system configurations allow for multiple GPUs to be connected to a single PCI-E bus and therefore this approach would help allieviate the pressure on the PCI-E bus. This will not dramatically affect the STR for most cases, therefore will not be a significant performance hit.
- **Reconstruction Size:** The minimum requirement for this application in the task partitioning phase is that at least one x-ray subimage and one volume slice fit on the device memory. It is possible for the reconstruction configuration to be awkwardly sized for a particular memory configuration of a GPU. A simple example would be a GPU with 2GB of available memory with a reconstruction task of one 1 GB slice with a 700 MB x-ray subimage; although this would easily fit within the 2GB limit, there is still hundreds of megabytes available but yet not enough to allow for an extra subimage and/or volume slice.
- **Tail-End of Reconstruction:** If the remaining work left to be performed by the GPU is much smaller than what the GPU is capable of computing at once, the resource maximization requirement would load extra x-ray images onto the device memory. This will only occur once per reconstruction and has very little impact on large reconstuctions.

Algorithm 1 gives a general description to the dynamic task partitioning as well the the kernel launch approach to the reconstruction of a given slice block. With the exception of Step 5a, all steps are performed by the CPU. Algorithm 1 executes independently for each GPU present on the system with the only atomic operation occuring at step 8. No synchronization between CPU threads (or between GPU threads for that matter) is necessary allowing for maximum performance.

Algorithm 2 describes the layout of the kernel computation for a given slice block and x-ray image subset where the ordering of the loops provide improved cache-hit rates as well as global memory traffic.

Algorithm 1 Dynamic Determination of GPU Task

while Reconstruction task queue not complete **do**
Step 1: Query GPU memory resources available
Step 2: Determine task partitioning given GPU memory resources and slice-to-texture ratio
Step 3: Allocate/Initialize Memory resources on CPU and GPU
Step 4: Upload reconstruction geometry information to GPU constant memory
Step 5:
for all image subsets **do**
-Upload image subset data from host to device texture memory
-Upon completion of upload, execute *(a)* and *(b)* simultaneously:
(a) GPU: Update slice block with image subset information via FDK kernel
(b) CPU: Prefetch next image subset. If last subset, free pinned-memory.
-Synchronize tasks *(a)* and *(b)*
end for
Step 6: Download slice block voxel information to host (storage optional)
Step 7: Free GPU memory resources
Step 8: Update reconstruction task queue
end while

Algorithm 2 FDK Kernel Layout

-Get thread id and voxel positions $p_1, \ldots, p_s$ based on id
if Thread id position within ROI **then**
for Every slice j in slice block **do**
-Set register value to zero
for Every image i in image subset **do**
-Determine texture interpolation coordinate within image i
-Update register value with texture fetch and scaling information
end for
-Update voxel p_j in global memory with register value
end for
end if

IV. Evaluation

The experiments were performed on a high-end workstation that consists of dual hexacore Intel Xeon X5690 processors clocked at 3.46GHz with hyper-threading for a total of 24 virtual CPU cores, 192 GB RAM and 2 Nvidia S2090 devices connected via 4 PCI-E 2.0 x16 host interface cards. Each S2090 unit contains 4 Tesla M2090 GPUs with 6 GB of GDDR5 memory apiece.

Each M2090 GPU contains 16 streaming multiprocessors (SM) that share a common L2 cache of 768 KB. The L2 cache services all load, store, and texture operations. Each SM contains 32 compute cores, 48 KB L1 cache, 8 KB constant memory cache, and 8 KB texture cache. Note that for the M2090, the L1 cache and shared memory are configurable to different sizes that can be determined by the user at compile time. The L1 cache was maximized in this work (thus minimizing shared memory) as shared memory was not utilized for

the reconstruction algorithm.

Timers used to calculate voxel processing throughput are CPU-based and include the time needed for all memory transfers, kernel launches, and prefetching operations necessary to completely reconstruct the given voxel subvolume assigned to the task. Voxel throughput was measured using two datasets; the first is $4000 \times 4000 \times 4000$ voxel (64 Gigavoxels) volume reconstructed from 1800 16-megapixel x-ray projection images where the measurements were taken about the center 2000 slices, the second is $10000 \times 10000 \times 10000$ voxel (1 Teravoxel) volume reconstructed from 10000 100-megapixel x-ray projection images where the measurements were taken on the center 100 slices of the volume. Measurements were taken for both datasets using 1 GPU and 8 GPUs. The kernels were compiled using CUDA version 4.1 and the cpu-based code was written in C++ using the Visual Studio 2008 C++ compiler.

The various cache hit-rates were measured using Nvidia's performance evalutation tool NSight. Kernel performance was measured on a single x-ray image subset using the 64-Gigavoxel dataset. Cache hit-rates could not be measured on the 1 teravoxel dataset due to NSight software limitations.

V. Results

Figure 1 illustrates voxel processing throughput of various subvolumes in the 64 gigavoxel dataset for various given slice-to-texture ratios (STR). The plot on the left shows that for one GPU, voxel throughput clearly benefits from small STR values with throughput peaking at $STR \approx 1.8$ with a throughput of 17.5 megavoxels per second. The average voxel throughput for $STR \leq 10$ is 15.84 megavoxels per second and the average voxel throughput for STR values greater than 10 was 10.21 megavoxels. The plot on the right side of figure 1 shows voxel throughput for a various subvolumes on an 8 GPU system. On average, voxel throughput still benefits from smaller STR values with an average voxel throughput of 13.11 megavoxels per second for $STR \leq 10$; for $STR > 10$, voxel throughput dramatically decreases to 4.54 megavoxels per second for a subvolume. Note that for the 8 GPUs case that with large STR values the computation time for a subvolume is highly variable when compared to smaller STR values so the average values are not necessarily representative of typical performance. It is likely that the observed variance in datapoints in figures 1 and 2 are due to various systems sources such as thread context switching, PCI-E bus pressure, and the GPUs themselves.

Figure 2 shows voxel processing throughput for subvolumes of the teravoxel dataset using various STR values. Since this dataset is extremely large (both in projections and volume), fewer STR values could be realized and therefore results are not as dramatic in throughput as with the 64 gigavoxel dataset but are still significant. For 1 GPU (left plot on figure 2), voxel throughput peaks at just under 0.51 megavoxels per second for a STR value of 0.52. For $STR \leq 0.9$, average throughput was about 0.497 megavoxels per second and 0.471 megavoxels per second for STR values greater than 0.9. On a system with 8 GPUs (right plot on figure 2), the performance is more variable but performance differences can still be observed with respect to STR size. The average throughput for $STR \leq 0.9$ was 0.46 megavoxels per second and 0.40 megavoxels per second with STR values greater than 0.9.

Figure 3 shows various GPU cache hit-rate performances on the 64 gigavoxel dataset for the reconstruction kernel. The upper plot of figure 3 shows the L1 cache hit-rate performance for various STR values. For this application, L1 cache is mostly populated with voxel values as well as a few kernel input variables that are used to determine loop length. As mentioned earlier, voxel values are only updated once per kernel launch. Regardless, the L1 cache hit-rate peaks for small STR values at 2.1% and decreases to 0.1% for STR values greater than 7.

The lower plot on figure 3 shows L2 and texture cache hit-rate performance. Although the texture cache hit-rate does not vary much with varying STR, it does peak at 70.4% for the smallest achieveable STR of about 0.47. The L2 cache clearly suffers from larger STR values, for STR values less than 1, L2 hit-rates are between 75 and 60% and as low as 10% for an STR value of 10.

VI. Conclusion

When viewed from a traditional approach, CT reconstruction is not an irregular problem and has excellent spatial locality. However, when utilizing GPU technology, one can lose spatial locality if the reconstruction is large and entire subvolumes are simulaneously reconstructed. This is due to the geometrical configuration of the imaging system and the unpredicable thread execution ordering. This work has shown that a CT reconstruction algorithm for GPUs can clearly benefit from an irregular approach for large-scale datasets by prefetching small batches of x-ray projection data and launching many kernels. This approach increases voxel throughput when compared to a partitioning method that only seeks to minimize data transfer uploads and kernel launches as is the common practice when creating GPU-based algorithms.

The main goal was to improve cache hit-rates to improve kernel performance. Utilizing texture cache exclusively for x-ray projection data as well as utilizing hardware-based interpolation improves computational performance dramatically but performance is

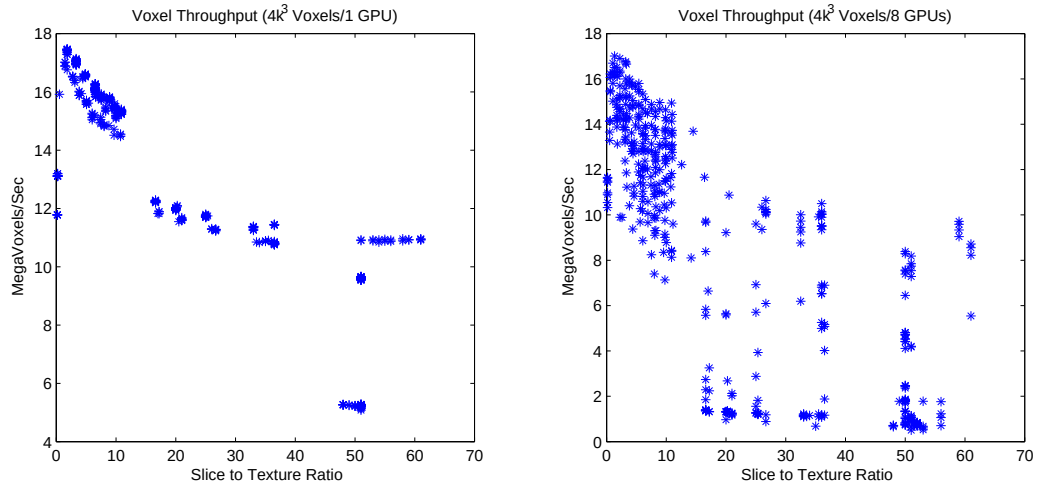

Fig. 1. Left: Reconstructed Voxel Throughput for 64 Gigavoxel Dataset using 1GPU, Right: Throughput using 8 GPUs

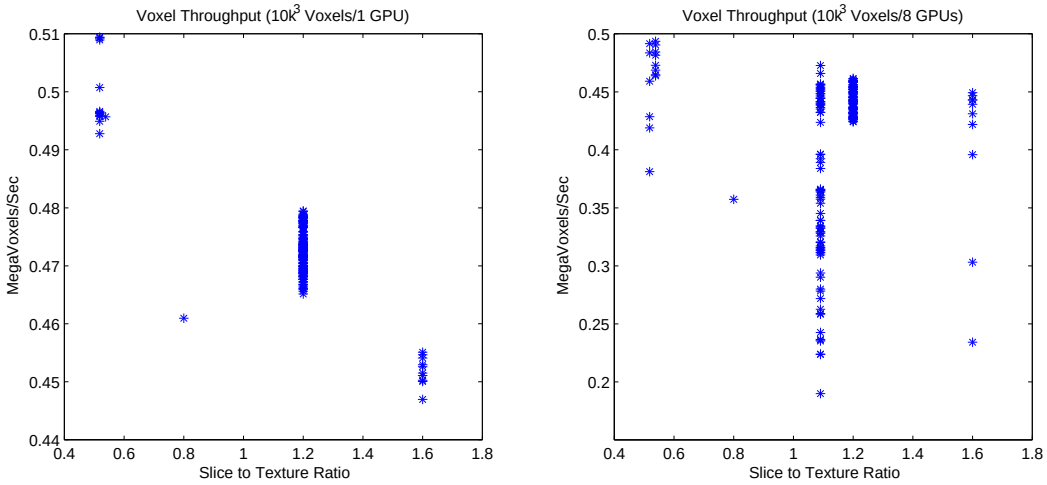

Fig. 2. Left: Reconstructed Voxel Throughput for Teravoxel Dataset using 1GPU, Right: Throughput using 8 GPUs

highly dependent on the cache hit-rates. This method can be used for large and small CT reconstruction tasks and maximizes performance for arbitrarily sized datasets. This work shows that smaller STR values are preferrable over larger values; however, arbitrary STR values are not acheivable due to various geometrical configurations as well as varying datasets sizes. It is possible to contrive a diabolical set in which only larger STR values ($\geq$ 1) are possible, however has not seemed to appear frequently in practice. In practice, maximizing the slice block in device memory tends to create the lowest STR values.

This work did not present any comparisons of "medical scale" datasets to other systems since much of the literature on GPU-based medical CT tended to either use older generation GPU hardware or the algorithms implemented were not the traditional FDK algorithm used for this work (much of the medical datasets are helical scans). Our algorithm is able to reconstruct sub-gigabyte datasets ($\approx 800^3$ voxels using 720 projections) at a rate of 30 slices per second per GPU.

For the general GPGPU community, this work has shown that regular CPU algorithms that are ported over to GPU environments may not result in an regular GPU algorithm. Although it is generally recognized that porting software for GPGPU application does not guarantee optimal performance, much of the literature suggests broad recommendations, such as minimizing memory transfers, when in fact one should consider possibly entirely different approaches that may have previously considered inefficient for CPU-based environments.

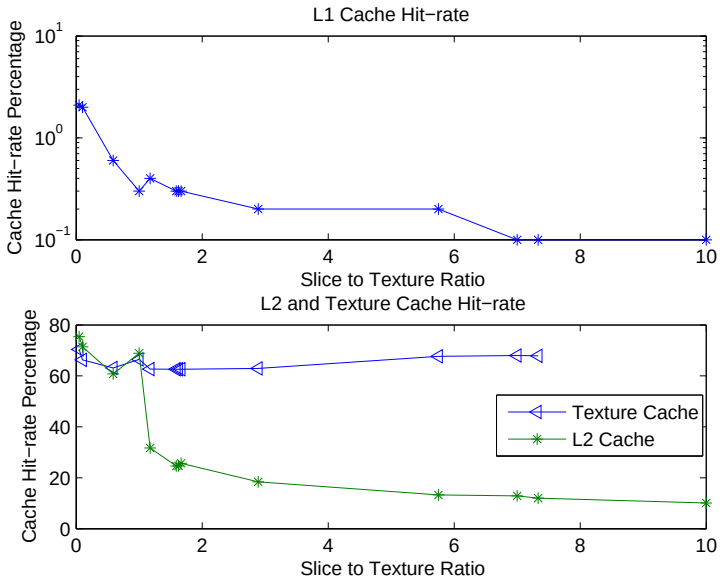

Fig. 3. Upper: L1 Cache hit-rate for reconstruction kernel. Lower: L2 and Texture Cache hit-rate for reconstruction kernel

VII. Acknowledgements

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

References

[1] H. H. Barrett and K. J. Myers, *Foundations of Image Science*. Wiley-Interscience, 2004.

[2] S. Izumi, S. Kamata, K. Satoh, and H. Miyai, "High energy x-ray computed tomography for industrial applications," *Nuclear Science, IEEE Transactions on*, vol. 40, no. 2, pp. 158 –161, apr 1993.

[3] L. Feldkamp, L. Davis, and J. Kress, "Practical cone-beam algorithm," *Journal of the Optical Society of America A*, vol. 1, no. 6, pp. 612–619, 1984.

[4] F. Xu and K. Mueller, "Ultra-fast 3d filtered backprojection on commodity graphics hardware," in *Biomedical Imaging: Nano to Macro, 2004. IEEE International Symposium on*, april 2004, pp. 571 – 574 Vol. 1.

[5] S. Xiao, Y. Bresler, and J. Munson, D.C., "Fast feldkamp algorithm for cone-beam computer tomography," in *Image Processing, 2003. ICIP 2003. Proceedings. 2003 International Conference on*, vol. 2, sept. 2003, pp. II – 819–22 vol.3.

[6] C. Axelsson and P. Danielsson, "Three-dimensional reconstruction from cone-beam data in o(n 3 logn) time," *Physics in Medicine and Biology*, vol. 39, no. 3, p. 477, 1994. [Online]. Available: http://stacks.iop.org/0031-9155/39/i=3/a=013

[7] F. Xu and K. Mueller, "Accelerating popular tomographic reconstruction algorithms on commodity pc graphics hardware," *Nuclear Science, IEEE Transactions on*, vol. 52, no. 3, pp. 654 – 663, june 2005.

[8] J. Sanders and E. Kandrot, *CUDA By Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley, 2011.

[9] W. mei W. Hwu, Ed., *GPU Computing Gems - Emerald Edition*. Morgan Kaufmann, 2011.

[10] F. Xu and K. Mueller, "Real-time 3d computed tomographic reconstruction using commodity graphics hardware," *Physics in Medicine and Biology*, vol. 52, no. 12, pp. 3405–3419, 2007.

[11] N. Corporation, *CUDA C Programming Guide v5.0*. http://www.nvidia.com, 2012.

[12] T. Mowry and A. Gupta, "Tolerating latency through software-controlled prefetching in shared-memory multiprocessors," *Journal of Parallel and Distributed Computing*, vol. 12, no. 2, pp. 87 – 106, 1991. [Online]. Available: www.sciencedirect.com/science/article/pii/074373159190014Z

[13] M. D. Lam, E. E. Rothberg, and M. E. Wolf, "The cache performance and optimizations of blocked algorithms," in *Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS-IV. New York, NY, USA: ACM, 1991, pp. 63–74. [Online]. Available: http://doi.acm.org/10.1145/106972.106981

[14] N. Corporation, *Nvidia CUDA C Best Practices Guide v5.0*. http://www.nvidia.com, 2012.