SANDIA REPORT

SAND2013-8059

Unlimited Release

Printed September 2013

High Performance Graphics Processor based Computed Tomography Reconstruction Algorithms for Nuclear and Other Large Scale Applications

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

Prepared by

Sandia National Laboratories

Albuquerque, New Mexico 87185 and Livermore, California 94550

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.

Approved for public release; further dissemination unlimited.

Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

NOTICE: This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government, nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, make any warranty, express or implied, or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represent that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof, or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof, or any of their contractors.

Printed in the United States of America. This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: <http://www.osti.gov/bridge>

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: <http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online>

High Performance Graphics Processor based Computed Tomography Reconstruction Algorithms for Nuclear and Other Large Scale Applications

Edward S. Jimenez

Abstract

The goal of this work is to develop a fast computed tomography (CT) reconstruction algorithm based on graphics processing units (GPU) that achieves significant improvement over traditional central processing unit (CPU) based implementations. The main challenge in developing a CT algorithm that is capable of handling very large datasets is parallelizing the algorithm in such a way that data transfer does not hinder performance of the reconstruction algorithm. General Purpose Graphics Processing (GPGPU) is a new technology that the Science and Technology (S&T) community is starting to adopt in many fields where CPU-based computing is the norm. GPGPU programming requires a new approach to algorithm development that utilizes massively multi-threaded environments. Multi-threaded algorithms in general are difficult to optimize since performance bottlenecks occur that are non-existent in single-threaded algorithms such as memory latencies. If an efficient GPU-based CT reconstruction algorithm can be developed; computational times could be improved by a factor of 20. Additionally, cost benefits will be realized as commodity graphics hardware could potentially replace expensive supercomputers and high-end workstations. This project will take advantage of the CUDA programming environment and attempt to parallelize the task in such a way that multiple slices of the reconstruction volume are computed simultaneously. This work will also take advantage of the GPU memory by utilizing asynchronous memory transfers, GPU texture memory, and (when possible) pinned host memory so that the memory transfer bottleneck inherent to GPGPU is amortized. Additionally, this work will take advantage of GPU-specific hardware (i.e. fast texture memory, pixel-pipelines, hardware interpolators, and varying memory hierarchy) that will allow for additional performance improvements.

Acknowledgment

This work was funded under LDRD Project Number 158182 and Title "High Performance Graphics Processor based Computed Tomography Reconstruction Algorithms for Nuclear and Other Large Scale Applications".

Contents

Nomenclature	11
1 Introduction	13
Problem Statement	13
Creative and Innovative Nature of R&D	13
Technical Approach and Leading Edge Nature of Work	14
Technical Risk and Likelihood of Success	16
Efficiently handling I/O	16
GPU-based Algorithm Design	17
The CUDA Environment	17
Qualification of the Team to Perform This Work	17
2 An Irregular Approach To Computed Tomography	19
Introduction	19
Approach	20
Irregular Approach	21
Massive Parallelism	22
Texture memory/Texture cache	22
Constant Memory	22
Data Prefetching to Pinned-Memory	22
Dynamic Task Partitioning	23
Computation Ordering	23
Implementation	23

Evaluation	25
Results	26
Conclusion	28
3 Rethinking CT Reconstruction and GPGPU Computing	31
Introduction	31
CPU-Based Reconstruction and Porting to GPUs	31
Exploiting Massive Thread Environments Properly	33
GPU Hardware Interpolation	34
Register and GPU-Cache Optimization	35
GPU Memory Utilization	36
Register Optimization	36
Constant Memory	36
Device Global Memory Fetches	36
Final Evolution	37
Evaluation	37
Results	38
Conclusion	39
4 Reconstructing a Multi-Terabyte CT Dataset	45
Introduction	45
Approach	46
Implementation	47
GPU Textures	48
Data Partitioning	50
CPU Thread Allocation	50
Analysis	51

Evaluation	51
Results	53
Conclusion	53
5 Irregular CT on Multiple GPUs Improves Energy-Efficiency Metrics	57
Introduction	57
Energy Efficiency	58
Irregular Computed Tomography	58
Implementation	59
Evaluation	61
Results	62
Energy Consumption	62
Performance Per Watt	63
Energy-Delay Product	63
Conclusion	64
References	67
 Appendix	
 A Technology Advances	71
 B Publications and Presentations	73

List of Figures

2.1	Left: Reconstructed Voxel Throughput for 64 Gigavoxel Dataset using 1GPU, Right: Throughput using 8 GPUs	27
2.2	Left: Reconstructed Voxel Throughput for Teravoxel Dataset using 1GPU, Right: Throughput using 8 GPUs	28
2.3	Upper: L1 Cache hit-rate for reconstruction kernel. Lower: L2 and Texture Cache hit-rate for reconstruction kernel	29
3.1	Average kernel runtimes with respect to image plane index for the 4000^3 voxel volume. Error bars represent the minimum and maximum runtimes	40
3.2	Average kernel runtimes variance with respect to image plane index for the 4000^3 voxel volume.	41
3.3	Average kernel performance improvement with respect to image plane index for the 4000^3 voxel volume.	41
3.4	Average kernel runtimes with respect to image plane index for the center image planes of a 10000^3 voxel volume. Error bars represent the minimum and maximum runtimes	42
3.5	Average kernel runtimes variance with respect to image plane index for the Center image planes of a 10000^3 voxel volume.	43
3.6	Average kernel performance improvement with respect to image plane index for the center image planes of a 10000^3 voxel volume.	43
4.1	Non-Modular Approach (IP = Image Plane)	48
4.2	Modular Approach (IP = Image Plane)	49
4.3	Scalability of GPU implementations for Teravoxel reconstruction	54
5.1	Energy Consumption in kWh for the 64 gigavoxel reconstruction.	62
5.2	Energy Consumption in kWh for the teravoxel reconstruction. Note: vertical axis is presented in a log-scale	63
5.3	Performance per watt for the 64 gigavoxel reconstruction	64

5.4	Performance per watt for the teravoxel reconstruction	65
5.5	Energy-Delay Product for the 64 gigavoxel reconstruction	65
5.6	Energy-Delay Product for the teravoxel reconstruction	66

List of Tables

4.1	Performance results for all 10,000 image planes of Teravoxel reconstruction . .	53
-----	---	----

Nomenclature

CT Computed Tomography

GPU Graphics Processing Unit

CPU Central Processing Unit

Device The whole GPU Co-processing unit, this includes the device memory, communications bus, other hardware on the GPU card.

Irregular Problem Any problem that has an unpredictable data access pattern or that requires a large amount of communication.

Voxel Three-dimensional pixel, or the smallest resolvable observation in the 3D volumetric approximation.

Slice Co-planar set voxels perpendicular to the radiation detector plane.

Image Plane See *Slice*

Slice Block A consecutive set of Image Planes

Texture Memory Read-only region of memory on a device that is created, for the purpose of this work, to exploit texture cache and hardware interpolation.

CUDA Compute Unified Device Architecture, a parallel computing platform created by NVIDIA

MIMD Multiple Instruction - Multiple Data

Chapter 1

Introduction

Problem Statement

The goal of this work is to develop a fast computed tomography (CT) reconstruction algorithm based on graphics processing units (GPU) that achieves significant improvement over traditional central processing unit (CPU) based implementations. The main challenge in developing a CT algorithm that is capable of handling very large datasets is parallelizing the algorithm in such a way that data transfer does not hinder performance of the reconstruction algorithm. General Purpose Graphics Processing (GPGPU) is a new technology that the S&T community is starting to adopt in many fields where CPU-based computing is the norm. GPGPU programming requires a new approach to algorithm development that utilizes massively multi-threaded environments. Multi-threaded algorithms in general are difficult to optimize since performance bottlenecks occur that are non-existent in single-threaded algorithms such as memory latencies. If an efficient GPU-based CT reconstruction algorithm can be developed; computational times could be improved by a factor of 20. Additionally, cost benefits will be realized as commodity graphics hardware could potentially replace expensive supercomputers and high-end workstations.

Creative and Innovative Nature of R&D

Development of a fully optimized reconstruction algorithm may require a dramatic restructuring of the entire algorithm in order to achieve optimal performance. This project will take advantage of the CUDA programming environment and attempt to parallelize the task in such a way that multiple slices of the reconstruction volume are computed simultaneously. This work will also take advantage of the GPU memory by utilizing asynchronous memory transfers, GPU texture memory, and (when possible) pinned host memory so that the memory transfer bottleneck inherent to GPGPU is amortized. Additionally, this work will take advantage of GPU-specific hardware (i.e. fast texture memory, pixel-pipelines, hardware interpolators, and varying memory hierarchy) that will allow for additional performance improvements. Many sponsors are not willing to invest in GPGPU algorithms at this time because the technology is still in its infancy and development environments vary widely.

Technical Approach and Leading Edge Nature of Work

Computed tomography (CT) imaging is an imaging method where a three-dimensional image of the inside of an object is reconstructed from a large number of X-ray projection images taken around an axis of rotation. Large sized problems (greater than 5 GB), where the projection images are high-resolution or the volume to be reconstructed is high-resolution, or both, CT reconstructions require significant computation and, in many cases, expensive computing resources. Various methods have been investigated to speed up CT reconstructions including the use of supercomputers, dedicated hardware, and commodity graphics processing units (GPU). The GPU is the most cost effective and versatile. The GPU takes advantage of millions of computation threads run in parallel. This is significantly more than the 1-12 parallel threads of a typical high-end central processing unit (CPU). The GPU has the additional advantage that it communicates to the host CPU with a bus rather than a network. Due to the dramatically different architectural environment of the GPU when compared to a CPU environment, the PI would like to point out that implementation of a CT reconstructor will most likely differ dramatically in developmental approach compared to traditional algorithms. Therefore this work is not merely a port of CPU-based code, but rather an entirely new restructuring of the reconstruction task so that GPU-specific hardware such as texture memory, fast hardware interpolators, pixel pipelines, and fast memory hierarchies can be utilized to the advantage of the reconstruction algorithm. This work will start out by using the FDK reconstruction approach developed by Feldkamp, Davis, and Kress (JOSA A, Vol.1, Issue 6, 612-619(1984)) which is the standard method used for industrial CT reconstruction for non-sparse datasets. Additional approaches will be explored that are based on iterative approaches that can be used for sparse datasets. Iterative reconstructors for large datasets in CT imaging have always been prohibitively expensive due to the massive amounts of data necessary as well as the extremely burdensome computational time required to accomplish the task. The massively parallel environment of the GPU now makes this approach potentially approachable. The advantages of an iterative reconstruction algorithm include corrections for photon noise, multi-energy, and CT artifacts, better image quality, and use on sparse datasets. The goal of this work will be to develop a GPU-based CT reconstruction algorithm that realizes speed-up improvements over traditional CPU-based implementations of arbitrarily sized CT datasets. GPU-based CT reconstruction methods have been implemented for medical imaging applications such as the works of Xu and Mueller (Phys. Med. Biol. 52(2007) 3405-3419, Trans. Nucl. Sci. 2003, SPIE Electronic Imaging 2007) as well as the work of Okitsu, Ino, and Hagihari (Parallel Computing 36(2010) 129-141) and have shown computational speed up of 10-60 times compared to traditional CPU-based implementations. These implementations do not trivially apply to datasets generated by non-medical applications. In medical applications, the patients radiation dose must be considered thus limiting the exposure time to acquire data. This limitation results in low-resolution x-ray images and thus the overall dataset is small (1 GB). These datasets can typically be processed in memory on the GPU or Host memory thus data management is not a significant performance bottleneck. In non-medical applications, radiation dose is typically not as big of a concern allowing larger higher resolution projections to be taken resulting in datasets that can be up to 1000 times larger (1 TB). Currently, only very large supercomputers can

process a 1 TB dataset in a reasonable amount of time, but even then analysis can still take days to complete. Using the medical implementations mentioned above would result in a large I/O memory and disk bottleneck because the dataset would not fit in the resident host memory or the GPU device memory, thus performance gains would not be realized. The FDK reconstruction algorithm is the main focus of this work to implement on the GPU. Exact three-dimensional reconstruction is not possible without a complete set of Radon data (continuous projection datasets) regardless of x-ray source trajectory as stipulated by the Tuy-Smith sufficiency condition. The FDK algorithm is a cone beam-based reconstruction method that approximates the exact reconstruction problem developed for technical investigation and provides an exact back projection instruction for the values measured in a discretized plane (Buzug, Computed Tomography). One of the features that make the FDK algorithm ideal for GPU implementation is that it is extremely parallalizable to a very fine-grain level which is ideal for the massively parallel environment of GPUs. Additionally, much of the imaging system geometry necessary for the computations can be stored in fast GPU caches that allow for greater efficiency. This work will implement a GPU-based CT algorithm that takes advantage of the following:

- Hundreds to millions of computational threads available on the GPU to perform the computationally expensive reconstruction.
- Multiple simultaneous slice reconstruction to maximize data reuse that will minimize host-to-device and device-to-host memory transfers.
- Asynchronous memory transfers that will allow the GPU to execute commands and data processing while data is being transferred between the host and device.
- Host pinned memory, if available, to maximize data upload and download speeds between host and device.
- Dynamic resource allocation to optimize code execution for any given GPU capable of running CUDA code. This will be accomplished by querying the GPU and evaluating the parameters of the CT reconstruction task to determine the optimal number of volume slices to reconstruct simultaneously for a given amount of device memory.
- Utilize multiple GPUs with OpenMP (Open Multi-Processing), when available, to add another level of parallelism to the reconstruction algorithm.
- GPU specific hardware that is not typically available to CPU algorithms that will allow for efficient execution of the algorithm as well as data transfer.

The approach outlined above will be successful because it will implement two levels of parallelism. The first level of parallelism is the reconstruction of multiple slices simultaneously on a single GPU. The second level of parallelism would be to partition a given reconstruction volume across multiple GPUs on a single host. Another ingredient for success in this work is an efficient data management protocol by utilizing the memory implementations listed above to amortize the memory and disk bottlenecks due to data transfer, reads, and writes. This work comes with four main challenges/concerns:

1. Efficiently handling disk and memory I/O
2. Developing a GPU-based algorithm that has traditionally been approached in a CPU-based context.
3. Using NVIDIAs Computed Unified Drive Architecture (CUDA) parallel programming environment.
4. Utilizing GPU-specific hardware for performance optimizations.

These issues have been acknowledged and are discussed in detail in section 2.3. It is expected that these issues can be overcome and should not hinder the development of this work and will be a valuable asset to Sandia National Laboratories. Advancements in the development of a GPU-based CT reconstruction algorithm would benefit Sandia National Laboratories and the weapons community as a whole. The potential impact of this work outside the laboratory and weapons community is enormous. Not only can this CT algorithm be used in the non-destructive evaluation field, but it could easily be utilized in the medical field, airport facilities, border and port-of-entry facilities as well as other national security applications that require reconstructions from projection data such as X-ray images. This work also provides a great opportunity for collaboration with Peter Smith and James Hunter of Los Alamos National Laboratory as they have a CPU-based reconstruction algorithm called RECON and could be a valuable resource in developing an efficient and usable algorithm. Peter and James are very well known throughout the Non-Destructive Testing field as leaders in CT algorithm development and testing. The GPU hardware and software architecture being investigated in this work will provide expertise in and motivation for an advanced computing platform that could be a precursor for Sandia National Laboratories next generation of supercomputers as well as a potential platform for future exascale computing architectures. Creating algorithms and methods that can efficiently utilize thousands of cores could be used to further understand how to develop algorithms that utilize billions of cores as Exascale computing dictates. This parallel computing approach mirrors Sandias future supercomputing development environment and approach. The expected results from this work are to have an efficient and accurate GPU-based reconstruction algorithm that can be run in either of two configurations with excellent performance:

1. Single Host / Single GPU
2. Single Host / Multiple GPUs

Technical Risk and Likelihood of Success

Efficiently handling I/O

The first risk associated with this work is the data management. Specifically, efficiently handling I/O at every level of the system (network, disk, memory, device memory, cache,

etc.). This will be dealt with by implementing asynchronous memory transfers and high data reuse by fine-grain multi-threaded parallelism. High data reuse will also minimize disk I/O.

GPU-based Algorithm Design

Developing a GPU-based algorithm that has traditionally been approached in a CPU-based environment is certain to be a risk in this work. This work will utilize Nvidia Corporations CUDA programming environment that allows easier access to GPU programming for non-graphics programmers. The PI for this project has a background in CUDA programming and algorithm development in a GPU context.

The CUDA Environment

Using a proprietary programming environment could potentially cause conflict for future supportability if the company was to stop supporting the environment. Since GPGPU programming has come into the mainstream, Nvidias CUDA programming environment has been by far the most popular, mature, and stable environment compared to the other alternatives for non-graphics programmers (such as OpenCL and DirectCompute). Nvidia has expressed enthusiasm about its programming environment and support is strong.

Qualification of the Team to Perform This Work

Edward Jimenez received his PhD in Applied Mathematics at the University of Arizona in 2010 and has extensive knowledge in mathematical modeling, optimization, multi-threaded and high-performance computing. Additionally, Edward was a student intern at SNL (Org. 1522 - Experimental Dynamics/Non-Destructive Evaluation and Model Validation) for 4 years in various CT-related projects under the guidance of Kyle Thompson. Kyle Thompson has been at SNL for over 25 years and is well known throughout the NDE and weapons community as an expert in CT imaging. Kyle will provide CT datasets as well as guidance in determining approaches to the proposed work.

Chapter 2

An Irregular Approach To Computed Tomography

Introduction

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.

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 [2]. Many industrial applications of Cone-beam Computed Tomography acquire a very large number (usually greater than 900) of x-ray projection images taken around an axis of rotation [10]. 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 $O(n^4)$ [8, 25]; work by Xiao et. al. [23] and Axelsson et. al [1] 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 com-

putational speed of CT reconstructions including the use of supercomputers, dedicated hardware (ASIC, FPGA, etc), and commodity graphics processing units (GPU) [24]. 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[22]. Driven mainly by the Gaming and CAD industry, investment in GPU-technology is unparalleled and has exhibited performance growth that triples Moore’s Law[24].

There has been extensive work done on applying GPU technology to various CT algorithms [17, 26, 25, 24] 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 simultaneously. 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.

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 alleviate this bottleneck is to minimize the total number of data transfers [4].

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 [22]. On the surface, this should guarantee minimal interruption during voxel processing.

The issue with the approach described above is two-fold. 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 utilizing read-only texture memory which utilizes texture cache and fast bilinear interpolation [17]. 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 computational efficiency as different threads in a warp could potentially be accessing x-ray data from different projection images.

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 [18]. Additionally, work done by Lam et. al. showed that cache interference in blocked algorithms can have a significant performance degradation for a given machine[15]. Overall, five aspects of the algorithm design, which uses the CUDA programming environment, will be addressed.

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 sub-volume. 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.

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 [17, 26, 25, 24]. 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.

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.

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 prefetching while the kernel is executing will guarantee that pauses between kernel launches are kept to a minimum.

Dynamic Task Partitioning

One desirable feature of this algorithm 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.

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 [5]. 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 register are being cached to the GPU global memory.

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 simultaneously 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 remaining memory after allocation will be utilized for additionally 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 alleviate 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 reconstructions.

Algorithm 3 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 3 executes independently for each GPU present on the system with the only atomic operation occurring at step 8. No synchronization between CPU threads (or between GPU threads for that matter) is necessary allowing for maximum performance.

Algorithm 4 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  $p_1, \dots, p_s$  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
```

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 evaluation 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.

Results

Figure 2.1 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 2.1 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 data-points in figures 2.1 and 2.2 are due to various systems sources such as thread context switching, PCI-E bus pressure, and the GPUs themselves.

Figure 2.2 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

Figure 2.1. Left: Reconstructed Voxel Throughput for 64 Gigavoxel Dataset using 1GPU, Right: Throughput using 8 GPUs

on figure 2.2), 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 2.2), 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 2.3 shows various GPU cache hit-rate performances on the 64 gigavoxel dataset for the reconstruction kernel. The upper plot of figure 2.3 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 2.3 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 achievable 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.

Figure 2.2. Left: Reconstructed Voxel Throughput for Teravoxel Dataset using 1GPU, Right: Throughput using 8 GPUs

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 simultaneously reconstructed. This is due to the geometrical configuration of the imaging system and the unpredictable 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 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 preferable over larger values; however, arbitrary *STR* values are not achievable 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 (≥ 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

Figure 2.3. Upper: L1 Cache hit-rate for reconstruction kernel. Lower: L2 and Texture Cache hit-rate for reconstruction kernel

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 a 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.

Chapter 3

Rethinking CT Reconstruction and GPGPU Computing

Introduction

In the previous chapter, claims were made of kernel optimizations without demonstrating to the reader how significant these optimizations can impact performance metrics. This chapter will focus on emphasizing the importance of kernel design and how this design differs from a traditional CPU-based design; this chapter will present the utilization of the massively multi-threaded environment of graphics processors (GPUs) to improve the computation time needed to reconstruct large computed tomography (CT) datasets and the arising challenges for system implementation. Intelligent algorithm design for massively multi-threaded graphics processors differs greatly from traditional CPU algorithm design. Although a brute force port of a CPU algorithm to a GPU kernel may yield non-trivial performance gains, further measurable gains could be achieved by designing the algorithm with consideration given to the computing architecture. Previous work has shown that CT reconstruction on GPUs becomes an irregular problem for large datasets (10GB-4TB)[12], thus memory bandwidth at the host and device levels becomes a significant bottleneck for industrial CT applications. We present a set of GPU reconstruction kernels that utilize various GPU-specific optimizations and measure performance impact.

CPU-Based Reconstruction and Porting to GPUs

The CUDA programming environment, as well as other GPU-programming languages (OpenCL, DirectCompute, etc.), have made GPGPU technology readily accessible to a large portion of the scientific computing community. Frequently, an honest first attempt to implement an algorithm on a GPU is to perform a brute force port of a CPU-based approach to a GPU-based implementation. It has been the experience of the authors that a blind CPU-to-GPU port of a properly parallelized algorithm will frequently yield a speedup in computation by a factor between $2\times$ and $6\times$. This type of performance gain is typically sufficient for many small-scale applications and thus the added effort to exploit the GPU-specific hardware does not benefit the user significantly.

A CPU-based reconstruction will typically loop over the projection data and iteratively update a single voxel on a given image plane. This process is repeated for every voxel on every image plane. For a multi-core CPU with n threads, the process is similar with the exception that n voxels are simultaneously updated. Currently, a typical system has an n that ranges from 2 to 32.

A simple GPU-ported kernel would allocate a computation thread for every voxel on an image plane and update each voxel in parallel with the given x-ray projection data iteratively. A CPU-based routine would loop over the image planes with a nested loop within iterating over projection data. Thus, for a reconstruction with N image planes and M projections, the kernel would be launched MN times.

Algorithm 3 CPU Kernel Launcher

Input: Projection Images $(P_1, P_2, \dots, P_M)$, Scan Geometry (G) , Voxels Per Image Plane (n)

Output: Voxelized Volume Reconstruction $I_1, I_2, \dots, I_N$

for Every image plane I_i **do**

 Allocate memory on GPU Device for I_i

 Initialize all values in GPU allocated array I_i to 0

for Every Projection Image P_j **do**

 Allocate memory on GPU Device for P_j

 Upload P_j to GPU Device

 Launch Ported Kernel with n GPU threads (see algorithm 4)

 Free P_j on GPU Device

end for

 Download I_i from GPU Device

 Free I_i on GPU Device

end for

Algorithm 4 Ported Kernel

Input: I_i, P_j, G

Output: I_i updated with data from P_j

 Get thread id and designated voxel in I_i

if Voxel in Region of Interest **then**

 Calculate back-projection path position, $\vec{b}$, within P_j

 Calculate bilinear interpolation weights, $\vec{w}$

 Calculate 2D interpolation on P_j based on $\vec{w}$ and $\vec{b}$

 Update designated voxel in I_i

end if

Algorithms 3 and 4 are a possible implementation of a ported version of reconstruction. The largest performance gain is realized from performing the bilinear interpolation for every voxel in parallel. The most undesirable traits of the implementation is the large amount of data uploads, downloads, and kernel launches required. Each of these operations has significant overhead that sacrifices performance [5].

Exploiting Massive Thread Environments Properly

Algorithms 3 and 4 are an example of a brute force implementation that requires minimal effort in porting over to a graphics processor. The next reasonable step would be to transplant the nested for-loop in algorithm 3 into the GPU kernel. Transferring the for-loops over to the kernel is desirable as this would reduce the number of kernel launches required to complete the reconstruction task as well as allowing the GPU to execute for longer periods of time which would improve the voxel processing throughput.

To implement the modification, one must still consider the almost arbitrary scan configurations and acquisition hardware. Thus, it is very likely that the entire projection dataset as well all imaging planes could not entirely reside on the device memory simultaneously. Therefore, a projection data and subset of image planes will instead be used where the projection block contains a subset of the relevant projection images and the image plane block will contain the image planes that will be processed simultaneously. For large-scale reconstructions, this implementation will iterate over all image plane blocks and projection data blocks.

Algorithm 5 CPU Kernel Launcher (Block Scheme)

Input: Projection Images $(P_1, P_2, \dots, P_M)$, Scan Geometry (G) , Voxels Per Image Plane (n)

Output: Voxelized Volume Reconstruction $I_1, I_2, \dots, I_N$

Determine blocking of projection data $(B_1^P, B_2^P, \dots, B_{M'}^P)$

Determine blocking of image blocks $(B_1^I, B_2^I, \dots, B_{N'}^I)$

for Every image block B_i^I **do**

 Allocate memory on GPU Device for B_i^I

 Initialize all values in GPU allocated array B_i^I to 0

for Every Projection image block B_j^P **do**

 Allocate memory on GPU Device for B_j^P

 Upload B_j^P to GPU Device

 Launch Blocked Kernel with n GPU threads (see algorithm 6)

 Free B_j^P on GPU Device

end for

 Download B_i^I from GPU Device

 Free B_i^I on GPU Device

end for

Algorithms 5 and 6 implement the blocking scheme discussed above, note that the CPU-based kernel launcher is essentially unchanged except for two-partitioning tasks and the nested for-loops now iterate over blocks of image planes (sub-volumes) and projection image blocks. Any blocking scheme can be used to accommodate any data format. The kernel presented in algorithm 6 will now take in blocked data, each of the n GPU threads launched will be assigned a subset of voxels to update with the projection data in B_j^P . Note that one could also launch more GPU threads on the GPU so that the outer for-loop in the kernel can be completely eliminated. We contend that for large-scale data, the performance difference

Algorithm 6 Blocked Kernel

Input: B_i^I, B_j^P, G **Output:** B_i^I updated with data from B_j^P Get thread id and designated voxel in each image plane in B_i^I **if** Voxels in Region of Interest **then** **for** Each Image Plane I_h in B_i^I **do** **for** Each Projection image P_k in B_j^P **do** Calculate back-projection path position, $\vec{b}$, within P_k Calculate bilinear interpolation weights, $\vec{w}$ Calculate 2D interpolation on P_k based on $\vec{w}$ and $\vec{b}$ Update designated voxel in I_h **end for** **end for****end if**

is likely negligible as the management of the increased number of GPU-threads becomes burdensome and GPU memory bus would be over-saturated.

GPU Hardware Interpolation

As mentioned earlier, the bilinear interpolation in the reconstruction algorithm is computationally expensive. One of the features of GPUs that differs from CPUs is hardware-based interpolation capabilities. The potential drawback is that for current technology, hardware-based interpolation is done in 24-bit precision [4] which differs from the 32- or 64-bit precision that is frequently used. Fortunately, many imaging and inspection applications only utilize 16-bit precision. Thus, as long as a numerically stable approach is implemented, then precision will remain adequate.

To utilize the interpolation hardware, the projection data must be uploaded to the GPU as a read-only texture array. For current state-of-the-art, utilizing texture arrays allows for the kernel to utilize fast texture cache on the GPU for a potential boost in performance. To accommodate multiple projection images, one could utilize either multiple textures, a large texture with tiled projection images, or layered textures [4, 5], depending on the particular device being utilized.

Algorithms 7 and 8 implement algorithms 5 and 6 except they exploit the hardware interpolation capabilities of the GPU. Note that in algorithm 7, the projection data block is now allocated outside of the nested for-loop. The texture array is updated and reused during reconstruction. This allows for a reduced number of memory allocation and deallocations. Algorithm 8 has been simplified by eliminating the calculations of the weights $\vec{w}$ and the 32- or 64-bit interpolation operation and replacing it with a single texture fetch.

Algorithm 7 CPU Kernel Launcher (HW Interpolation Scheme)

Input: Projection Images $(P_1, P_2, \dots, P_M)$, Scan Geometry (G) , Voxels Per Image Plane (n)

Output: Voxelized Volume Reconstruction $I_1, I_2, \dots, I_N$

Determine blocking of projection data $(B_1^P, B_2^P, \dots, B_{M'}^P)$

Determine blocking of image blocks $(B_1^I, B_2^I, \dots, B_{N'}^I)$

Allocate *texture* memory on GPU Device for largest $B_j^P \in \{B_1^P, B_2^P, \dots, B_{M'}^P\}$

for Every image block B_i^I **do**

Allocate memory on GPU Device for B_i^I

Initialize all values in GPU allocated array B_i^I to 0

for Every Projection image block B_j^P **do**

Upload B_j^P as a texture to GPU Device

Launch Blocked Kernel with n GPU threads (see algorithm 8)

Free texture B_j^P on GPU Device

end for

Download B_i^I from GPU Device

Free B_i^I on GPU Device

end for

Free Texture memory allocated on Device

Algorithm 8 HW Interpolation Kernel

Input: B_i^I, G

Output: B_i^I updated with data from texture B_j^P

Get thread id and designated voxel in each image plane in B_i^I

if Voxels in Region of Interest **then**

for Each Image Plane I_h in B_i^I **do**

for Each Projection image P_k in B_j^P **do**

Calculate back-projection path position, $\vec{b}$, within P_k

Perform texture fetch from P_k at position $\vec{b}$

Update designated voxel in I_h

end for

end for

end if

Register and GPU-Cache Optimization

Our final evolution of the reconstruction task is the implementation presented in Jimenez et al.[12]. This implementation involves improving algorithmic execution by minimizing wasted clock cycles on the device. To accomplish this, we exploit the following:

- GPU Memory Utilization
- Register Optimization

- Constant Memory
- Device Global Memory Fetches

Each will be addressed separately.

GPU Memory Utilization

Instead of uploading entire projection images, only relevant projection image data from a given projection block is uploaded. Determining relevant data is a trivial calculation and should have negligible effect on performance. The elimination of irrelevant projection data will improve the texture cache hit-rates as well as improve the utilization of device resources.

Register Optimization

When calculating $\vec{b}$, the order in which the calculations are executed as well as how many variables are utilized could potentially affect performance due to read-after-write dependencies and register pressure. Read-after-write dependencies have a latency of approximately 24 clock cycles for Nvidia GPUs [5]. Additionally, we implement a kernel such that register pressure is minimized as much as possible. This is achieved implicitly by the combination of optimizations in the list above.

Constant Memory

Until now, no mention of the parameters contained in $\vec{G}$ have been made. The vector $\vec{G}$ contains the scan geometry information (detector properties, source-to-detector distance, source-to-object distance, etc.). Throughout the reconstruction, $\vec{G}$ is fixed and is used in the calculation of $\vec{b}$. Therefore, we propose moving $\vec{G}$ into constant memory, which is a very fast user configurable cache that is disjoint from the L1- and L2-cache; thus reducing the traffic that goes through both cache levels.

Device Global Memory Fetches

When updating voxel information, memory bandwidth can be preserved by only updating voxel information when necessary. During iteration through projection data, a register will be updated, after these iterations complete, one voxel update is performed. The improvements are two-fold, not only is device bandwidth preserved for texture fetches, but also allows the L2-cache to more effectively feed the texture-cache.

Final Evolution

Algorithms 9 and 10 show an optimized implementation of the reconstruction task. Note that the kernel now only requires a single input due to the texture implementation of the projection data and the constant memory assignment of the scan geometry parameters. This implementation will reduce the number of wasted clock cycles by providing a memory access scheme that keeps each GPU multiprocessor occupied with work. The approach to avoid read-after-write dependencies will be highly variable in its implementation due to the different schemes for each particular reconstruction algorithm.

Algorithm 9 Final CPU Kernel Launcher Optimized

Input: Projection Images $(P_1, P_2, \dots, P_M)$, Scan Geometry $(\vec{G})$, Voxels Per Image Plane (n)

Output: Voxelized Volume Reconstruction $I_1, I_2, \dots, I_N$

Determine blocking of projection data $(B_1^P, B_2^P, \dots, B_{M'}^P)$

Determine blocking of image blocks $(B_1^I, B_2^I, \dots, B_{N'}^I)$

Upload $\vec{G}$ to GPU Constant Memory

for Every image block B_i^I **do**

 Allocate memory on GPU Device for B_i^I

 Initialize all values in GPU allocated array B_i^I to 0

 Allocate *texture* memory on GPU Device for largest *relevant* projection block needed

for Every Projection image block B_j^P **do**

 Determine Relevant Projection Data in $B_j^P, B_j'^P$

 Upload $B_j'^P$ as a texture to GPU Device

 Launch Kernel with n GPU threads (see algorithm 10)

end for

 Download B_i^I from GPU Device

 Free B_i^I on GPU Device

 Free texture on GPU Device

end for

Free constant memory

Evaluation

All kernel implementations were tested on a high-end single node workstation consisting of a Supermicro X9DRG-QF Motherboard, 512 GB DDR3 System memory, dual Intel Xeon E5-2687W octo-core processors clocked at 3.1 GHz with hyper-threading for a total of 32 CPU threads, and 2 Nvidia/Next-I/O Tesla S2090 Devices connected via 4 PCI-E 2.0 16x host interface cards. Each S2090 device consists of 4 Nvidia Tesla M2090 GPUs with 6GB of GDDR5 device memory and 16 multiprocessors each for a total of 2048 CUDA cores per

Algorithm 10 Final Kernel Optimized

Input: B_i^I **Output:** B_i^I updated with data from texture B_j^P Get thread id and designated voxel in each image plane in B_i^I **if** Voxels in Region of Interest **then** **for** Each Image Plane I_h in B_i^I **do**

Set register to zero

for Each Projection image P_k in B_j^P **do** Calculate back-projection path position, $\vec{b}$, within P_k while minimizing read-after-write dependencies Perform texture fetch from P_k at position $\vec{b}$

Update register with texture fetch data

end for Update designated voxel in I_h with register value **end for****end if**

S2090. For this work, only 1 M2090 GPU will be used to minimize performance influence from the host.

Each kernel's performance will be measured against two datasets; the first is a 4000^3 voxel volume reconstruction from 1800 16 mega-pixel (4000×4000) images, the second is a trillion voxel reconstruction from 10,000 100 mega-pixel ($10,000 \times 10,000$) images. All GPU kernels were written in CUDA (Version 5.0), host code was written in C++ using Microsoft Visual Studio 2008. Performance metrics consist of minimum, maximum, and average kernel runtime with respect to image plane position, as well as kernel runtime variance with respect to image plane position. As this work solely focuses on individual kernel runtime performance, other operations such as data transfers between host and device, and data transfers from host to storage media will not be measured.

Results

Figure 3.1 shows the average, minimum, and maximum kernel run-times for the 4000^3 for each implementation of the kernels. As expected, each evolutionary step of the kernel yields an incremental performance increase. Note that across all implementations, kernel runtime increases among the central image planes of the volume, this is due to more projection data being processed per kernel launch and is expected. In order to allow comparisons to the ported kernel implementation, all performance times were normalized with respect to the projection blocks used for the other three kernels. Figure 3.2 contains the variances of the kernel runtimes for each kernel. Notice that the fully optimized kernel (algorithm 10) is the only kernel with significantly lower variance, this implies that not only is algorithm 10 fast (figure 3.1), but also behaves more consistently across the entire reconstruction volume.

Figure 3.3 illustrates the incremental performance increase of the fully optimized version compared to all other implementations. We see that the overall performance improvement compared to the first implementation was almost a factor of 3.

Figure 3.4 displays the kernel runtimes for the center 100 image planes for the trillion voxel reconstruction. Runtimes are longer due to the increased computational expense. The results are still consistent with those observed in figure 3.1. The variance behavior of the trillion voxel dataset (shown in figure 3.5) is somewhat similar to that displayed for the 4000^3 dataset in that the fully optimized version exhibits the most consistent performance. Finally, figure 3.6 shows a slightly better performance improvement for the fully optimized kernel compared to the other kernels. This is due to the increased computational complexity and opportunity for parallelization. Also, the trillion voxel reconstruction will stress the limits of the hardware thus making discrepancies in performance more prevalent.

Conclusion

We have shown that GPUs can achieve a nontrivial increase in performance by properly utilizing GPU threads, specialized GPU hardware, and exploiting the unique cache structure to the advantage of the algorithm. This work has the potential to impact GPU applications in Green Computing, smart algorithm design, and high-performance computing. Many applications are not realizing the full effect of GPGPU technology, as this work is an example of the relatively minimal effort needed to redesign an algorithm and achieve significant performance gains. Many problems are arising where smart algorithm design will have a significant impact on power efficiency, performance, and computational time. This is especially true for new and emerging non-CPU computing architectures.

Figure 3.1. Average kernel runtimes with respect to image plane index for the 4000^3 voxel volume. Error bars represent the minimum and maximum runtimes

Figure 3.2. Average kernel runtimes variance with respect to image plane index for the 4000^3 voxel volume.

Figure 3.3. Average kernel performance improvement with respect to image plane index for the 4000^3 voxel volume.

Figure 3.4. Average kernel runtimes with respect to image plane index for the center image planes of a 10000^3 voxel volume. Error bars represent the minimum and maximum runtimes

Figure 3.5. Average kernel runtimes variance with respect to image plane index for the Center image planes of a 10000^3 voxel volume.

Figure 3.6. Average kernel performance improvement with respect to image plane index for the center image planes of a 10000^3 voxel volume.

Chapter 4

Reconstructing a Multi-Terabyte CT Dataset

Introduction

As stated in chapter 1, traditional CT reconstruction algorithms (both GPU- and CPU-based) focus on reconstructing small-scale medical datasets. Chapters 2 and 3 demonstrated that the traditional approaches are not sufficient to reconstruct large-scale CT datasets in a reasonable amount of time. This chapter will look ahead to "future-sized" datasets. The high-end for current industrial CT reconstruction tasks typically consist of reconstructing about 64 billion voxels from 1800 16-megapixel x-ray images. However, current scientific literature has identified several candidates for 100-megapixel camera; therefore, it may soon be possible to acquire 10,000 100-megapixel x-ray images as CT input to reconstruct a teravoxel (1 trillion voxels) volume. This chapter describes a modularized approach which supports a more efficient utilization of GPU resources and compares performance of this approach to various CPU- and GPU-based algorithms.

To reconstruct such a massive dataset, such as the one mentioned above, simply adding extra GPUs would not be an option as memory and storage bottlenecks would result in prolonged periods of GPU downtime, thus negating performance gains. Additionally, current reconstruction algorithms would not be sufficient due to the various bottlenecks in the processor hardware. Past work has shown that CT reconstruction is an irregular problem for large-scale datasets on a GPU due to the massively parallel environment. This work proposes a high-performance, multi-GPU, modularized approach to reconstruction where computation, memory transfers, and disk I/O are optimized to occur in parallel while accommodating the irregular nature of the computation kernel. Our approach utilizes a dynamic MIMD-type of architecture in a hybrid environment of CUDA and OpenMP. The modularized approach showed an improvement in load-balancing and performance such that a 1 trillion voxel volume was reconstructed from 10,000 100 megapixel projections in less than a day. While the reconstruction process is similar in industry and medicine, the datasets used in industry are around $4,000^3$ volumetric pixels (voxels) in size compared to the $128^3 - 1,000^3$ voxel volumes in the medical field because medical imaging must limit the patient's radiation exposure time. The larger datasets require more processing power and computation time and can require multiple days to complete even on a multi-core, multi-threaded computer. Fur-

ther, with improvements in Gigapixel camera architecture[6] and new advancements in high resolution cameras such as those by the European Space Agency[14] and DARPA [7], industrial dataset projections may soon be 100-megapixels ($10,000 \times 10,000$) in size, and, with our current reconstruction techniques, would not be possible to reconstruct in a reasonable amount of time.

Many efforts have been attempted to bridge the gap between the processing times of medical and industrial datasets by utilizing the massively parallel architecture of the Graphics Processor Unit (GPU)[21]. GPUs can provide an orders-of-magnitude performance gain on highly parallel problems such as CT reconstruction where the same mathematical operations are done on multiple voxels in parallel[17]. However, even with multiple GPUs, industrial reconstruction can still take hours to complete, and with larger datasets in the foreseeable future, the industrial imaging community needs greater processing power and capabilities to handle such datasets. The naïve solution of adding more GPUs would give diminishing returns on runtime improvement. This is due to hardware limitations in bandwidth and memory on both the host and GPU device, adding more GPUs will create a bandwidth bottleneck and result in prolonged periods of GPU idle time.

One approach described in chapter 2 improves performance on industrial size datasets, implements an irregular algorithm that exploits the GPU's unique cache structure and executes small X-ray data prefetches from the host to the device(s). While this approach, when using 8 GPUs on a 64 Gigavoxel dataset, roughly tripled voxel throughput compared to a more direct port of the CPU-based CT reconstruction to GPU kernels, it still experienced bottlenecks while transferring reconstructed slices from GPU memory onto storage media, which caused suboptimal GPU utilization since GPUs were idling while waiting for a memory transfer and storage media write to complete.

This paper will present a further improvement to the previously mentioned irregular algorithm that takes a modularized, MIMD approach to ameliorate the disk transfer bottleneck and maximize CPU and GPU resource utilization. It will then use the large-scale (1 Teravoxel), synthetic dataset from Ref. [13] to evaluate the algorithm's performance and the GPU scalability in order to predict this method's applicability to current and future industrial datasets.

Approach

As mentioned in 4, the approach from Ref. [13] suffered from prolonged GPU idle time when performing storage operations. The bottleneck was a consequence of stalling GPU computation while writing the reconstructed image planes to storage media. Although this freed memory for the next image plane sub-volume, it severely underutilized GPU resources. The approach outlined in this paper addresses the inefficiency by storing the image planes in host memory and writing them to storage media in parallel once a GPU has reconstructed its given sub-volume. After a GPU has transferred its image planes to the host, the next sub-

volume reconstruction is immediately executed, leading to drastically improved utilization of the GPU processor.

More specifically, dedicated threads are dynamically allocated on the host machine to exclusively handle either prefetching the X-ray images from storage media, controlling the GPUs, or transferring the images planes to storage media. Once the X-ray images are in host memory, the GPUs reconstruct sub-volume blocks by iteratively requesting X-ray images from host memory, computing the image planes, and transferring the reconstructed image planes to host memory. While the GPU controlling threads reconstruct sub-volumes, the data transfer threads write the image planes stored in host memory to storage media.

To accommodate environments with limited memory, the total volume is partitioned so that for each section, an appropriate number of X-ray images are prefetched so that all the X-ray images and reconstructed image planes for that section can fit into host memory. Due to the varying number of X-ray images needed to reconstruct a single slice throughout the volume, the number of X-ray images must be dynamically assigned during reconstruction. Additionally, the GPUs treat each volume block as a total volume and iteratively reconstruct sub-volumes of that block. Once all the reconstructed image planes for a block are written to storage media, the host memory is freed and the next batch of X-ray images are read into host memory for the next volume block. By partitioning the volume to accommodate a variety of systems, the process is not limited to systems with a massive amount of free host memory. Note that for industrial applications, if the total volume was entirely stored on host memory, the memory requirements would be massive, on the order of 100 Gigabytes to several Terabytes.

Implementation

At a high level, this algorithm runs in two main loops, an outer loop for partitioning the entire volume into sub-volumes and an inner loop for iterating through each sub-volume and executing the GPU kernels. The size of each sub-volume is dynamically determined by maximizing the number of X-ray images that can be prefetched while maintaining that all relevant X-ray images and reconstructed image planes fit in host memory.

For a given sub-volume, once the X-ray images have been read into host memory from storage, host threads are allocated to be either GPU controlling or write threads, which allows for GPU execution and image plane transfer from host memory to storage media to occur simultaneously compared to the processes being disjoint in the method described in Ref. [13]. Figure 4.1 and Figure 4.2 give a visual representation of how the parallelization affects execution. Additionally, as first described in Ref. [13], GPU textures are fully exploited by using small sets of X-ray images to be prefetched from host memory to device memory. The inner loop finishes once the sub-volume has been successfully written to storage by the CPU threads, and the outer loop finishes once the entire volume has been reconstructed (see Algorithm 1 for a high-level description).

Figure 4.1. Non-Modular Approach (IP = Image Plane)

The key irregular features of this approach are its utilization of GPU textures, dynamic partitioning of X-ray images to fit into available host memory, and CPU thread allocation to parallelize computation and image plane transfer to storage media.

GPU Textures

This feature follows the non-modular implementation from Ref. [13]. The fast texture cache hit rate of the GPUs is exploited by dynamically fitting the number of X-ray images to transfer to the GPU to a user inputted slice-to-texture ratio. Setting the ratio low (between 0 and 1) implies most of the GPU memory is reserved for storing finished image planes and the small amount left for X-ray images fits into the texture cache. Although the GPU needs to

Figure 4.2. Modular Approach (IP = Image Plane)

transfer more X-ray images to and from its texture cache to reconstruct all the image planes, its runtime improves because of a high texture cache hit rate along with an improved L1, L2, and constant cache hit rate. Overall, performance improves by going against standard recommendations to maximize data transfer from the host to the GPU[5].

Data Partitioning

In order to accommodate not only workstations with different amounts of system memory but also the varying number of voxels to reconstruct and differing acquisition geometries, the number of image planes to reconstruct in the inner loop must be dynamically determined. The challenge in this determination is in the irregular number of X-ray images needed to reconstruct a given image plane throughout the volume. In order to maximize performance, a constant number of image planes per iteration cannot be used.

At the beginning of each outer loop iteration, the number of image planes to reconstruct is determined by calculating the host memory required to reconstruct all remaining image planes (the memory must hold the image planes and necessary X-ray images). If the host memory required is greater than 90 percent of the free memory on the system, the number of image planes to reconstruct is decremented by one, and the required memory is recalculated. This process is repeated until the required host memory is less than or equal to 90 percent of the free host memory. The 90 percent value is used versus 100 percent to account for any inaccuracies in the calculations and to allow space for OS-specific requirements. Although this linear search for the maximum number of image planes is not of optimal efficiency, the extra time to run the search is negligible when compared to the overall performance of the algorithm.

Once the maximum number of image planes to reconstruct is calculated, CPU threads are initialized to prefetch the required X-ray images into host memory. Once all X-ray images are stored in CPU memory, the inner loop begins the reconstruction on that section of image planes. This process repeats until the entire volume is reconstructed.

CPU Thread Allocation

As detailed above, once the X-ray images necessary for reconstructing a sub-volume are in system memory, the inner loop begins reconstruction. However, before the inner loop starts, CPU threads are assigned to be either GPU controlling threads or image plane transfer threads. Since the number of GPUs available on the system is known a priori, one CPU thread is assigned to each GPU.¹ Then, the remaining number of CPU threads on the system (total number of CPU threads minus number of GPUs) are assigned to write the

¹If there are not as many CPU threads as GPUs on the system, one CPU thread is still assigned to each GPU and the operating system's thread scheduler is expected to accommodate the excess CPU threads

reconstructed image planes from host memory to storage.²

The CPU threads assigned to each GPU communicate to the CPU write threads that an image plane is ready to be written to disk through an integer array the size of the number of image planes being reconstructed in a given inner loop.³ Once a GPU is finished reconstructing, the CPU thread allocated to it sets the associated cells in the communication array to 1. Each CPU write thread continually loops through a disjoint subset of the array's cells to avoid memory collisions and the same image plane being written to storage twice, checking if any cell is 1. If a cell is 1, then that CPU thread sets that cell back to 0 and writes the associated image plane to storage media. Once all the image planes from a sub-volume are transferred to storage, the inner loop ends, host memory is freed, and the outer loop repeats.

This scheme of dynamic task partitioning allows for modularity between the GPU reconstruction and image plane writing. Further, the number of CPU threads used is determined by the limitations of the system, meaning a wider variety of workstations can run this algorithm.

Analysis

Evaluation

The experiments were performed on a workstation running Windows Server 2008 consisting of 2 Intel(R) Xeon(R) E5-2687W octo-core processors clocked at 3.1 GHz with hyper-threading for a total of 16 CPU cores, 32 threads, 512 GB RAM, and 2 Nvidia S2090 devices connected via 4 PCI-E 2.0 x 16 host interface cards. Each S2090 contains 4 Tesla M2090 GPUs where each GPU has 6 GB of GDDR5 memory and 16 multiprocessors where each multiprocessor has 32 CUDA cores. The workstation has an eight drive Raid 0 array managed by an Intel RAID Controller (RS25AB080) with 1 GB of DDR3 Cache. The drives in the array are 3TB Hitachi 7.2K RPM SATA 6Gb/s Drives.

The CPU-based time collected measured total reconstruction time, which included all memory transfers, GPU kernel launches, and disk I/O operations. The time was measured for the non-modular GPU version (version A), the dynamic memory allocation and parallelized disk I/O GPU version (version B), a OpenMP parallel CPU-based version (version C), and a single thread CPU-based version (version D). All were run using a 10,000 x 10,000 x 10,000 voxel (1 Teravoxel) volume reconstructed from 10,000 100-megapixel x-ray projection images. Versions A and B were run using 1, 2, 4, 6, and 8 GPUs. The measurements for the 1 and 2 GPU runs were taken on the first 200 and the middle 200 image planes and then averaged. The other GPU runs' measurements were taken on all 10,000 image planes. For version C and D, due to the extremely long run time per single image plane (about 0.4 hours

²If there are more GPUs than CPU threads, one CPU thread is assigned to be the transfer thread

³Determined as described in 4

Algorithm 11

```
numWriteThreads = totalNumThreads − numGPUs
worldSliceDone = 0
while worldSliceDone ≤ worldTotalSlices do
  determine totalSlices so X-ray images and output fit in memory
  slicesDone = 0
  while sliceDone < totalSlices do
    read in X-ray images {once per outer loop}
    spawn threads with ids from 0 to totalNumThreads − 1
    if thread's id < numGPUs then
      assign thread to GPU
      run GPU compute kernel
      transfer slices to CPU
      update write array
    else
      numWrittenSlices = 0
      while numWrittenSlices < totalSlices do
        for cell in subset of write array do
          if cell is 1 then
            set cell to 0
            write that image plane from CPU memory to storage
          end if
        end for
      end while
    end if
  end while
  synchronize CPU threads
  worldSliceDone += slicesDone
end while
```

for C and 0.45 hours for D), the measurements were only taken on image planes 300 to 303, averaged, and then extrapolated for the full 10,000 image planes.

Lastly, the GPU kernels were compiled using CUDA version 5.0, and the CPU code was written in C++ using Visual Studio 2008's C++ compiler.

Results

Figure 4.3 illustrates voxel processing throughput of the modular GPU version (version B) compared to the non-modular version (version A) of the Teravoxel reconstruction using 1, 2, 4, 6, and 8 GPUs. For a single GPU, version B's voxel throughput is not noticeably improved compared to A's (0.27 versus 0.24) because the host system is able to keep up with the memory transfers and storage I/O required, meaning the GPU does not experience significant idle time in either version. As the number of GPUs increases, the GPUs in version A are forced to wait longer to begin processing the next set of X-ray images because of the increased demand on limited storage I/O. Due to version B's storage I/O modularization and parallel GPU kernel launches, it better handles a higher number of GPUs and has improved scalability with respect to the number of GPUs.

Table 4.1 displays the overall runtime of the 4 versions, including memory transfers, disk I/O, and GPU kernel launches. The rightmost column displays the speedup of that version compared to the single thread CPU-based version. The GPU versions clearly outperform the CPU-based versions with the modular implementation being 2,260 times faster than the single thread version.

Algorithm	Time(hours)	Speedup
Single Thread CPU (C)	44,987	N/A
OpenMP CPU (D)	4,039	11.1
Non-Modular 8 GPU (A)	23.3	1,927
Modular 8 GPU (B)	19.9	2,260

Table 4.1. Performance results for all 10,000 image planes of Teravoxel reconstruction

Conclusion

For CT reconstructions in the industrial field, utilizing the massively parallel architecture of the GPU can dramatically improve performance. This orders-of-magnitude speedup is important for industrial applications since the processing times for their larger datasets

Figure 4.3. Scalability of GPU implementations for Teravoxel reconstruction

(on the order of Gigavoxels) using a CPU-based algorithm can take days to complete. The algorithm described in Ref. [13] is one such GPU-based CT reconstruction which goes beyond the direct port of a CPU-based algorithm to GPU kernels and improves performance even more by taking advantage of the GPU-specific capabilities and hardware.

That implementation, however, does not fully utilize the GPU due to a non-optimal memory transfer and storage media writing scheme. While this inefficiency may not seem significant for current reconstructions (Gigavoxels in size), it will noticeably hurt performance on future-sized datasets (Teravoxels in size). These datasets are soon to be realized due to the advancements in high performances cameras, and the industrial community needs a CT reconstruction technique that can handle such massive amounts of data.

The implementation described in this paper modifies the one from Ref. [13] to dynamically modularize the memory transfers and parallelize the GPU computations and storage media writing. These improvements cause it to outperform the non-modular version on a future-sized dataset 1 Teravoxel in size.

This work has shown that transforming a CPU-based algorithm into one using GPUs involves not only rethinking the nature of the computations but also restructuring the memory and storage media usage. Future improvements to this implementation include expanding it to a cluster implementation where each node has access to at least one GPU. The cluster version will also need to dynamically determine how many image planes each node should reconstruct in a way that takes into account not only the memory limits of the node but also the number and type of GPU. This and future work will help prepare the industrial community for the technological advancements to come in CT reconstruction.

Chapter 5

Irregular CT on Multiple GPUs Improves Energy-Efficiency Metrics

Introduction

This chapter will investigate energy-efficiency for various real-world industrial computed-tomography reconstruction algorithms, both CPU- and GPU-based implementations. Many of the commercial advertising for graphics processors claim that GPUs have a high performance-to-watt ratio. Unfortunately, finding verification of this claim in the scientific literature proved to be a challenge. Additionally, the irregular nature of CT reconstruction algorithms adds another level of complexity to the characteristics of energy metrics due to the unique bottlenecks and memory access patterns. We show that the energy required for a given reconstruction is based on performance and problem size. There are many ways to describe performance and energy efficiency, thus we will investigate multiple metrics including performance-per-watt, energy-delay product, and energy consumption. We found that irregular GPU-based approaches [12] realized tremendous savings in energy consumption when compared to CPU implementations while also significantly improving the performance-per-watt and energy-delay product metrics. Additional energy savings and other metric improvement was realized on the GPU-based reconstructions by improving storage I/O by implementing a parallel MIMD-like modularization of the compute and I/O tasks.

Due to the enormous input datasets (usually between 1 gigabyte and 20 gigabytes) and the computational complexity, reconstruction of industrial datasets can easily consume a significant amount of energy. Therefore, consideration should be taken in choosing an appropriate computing platform. This work will evaluate multiple implementations of the standard Feldkamp-David-Kress (FDK) reconstruction algorithm [8] that is routinely used for non-destructive testing and industrial scale applications. The evaluation will include multiple energy-efficiency metrics, including energy consumption, performance-per-watt, and the energy-delay product to obtain a broad understanding of the energy-efficiency characteristics of this real-world application.

Energy Efficiency

A very simple way of looking at energy efficiency is to measure the total energy used by each system. The less energy used by one system in comparison to another, the better. However, this misses the very important dimension of performance. Two systems may expend the same amount of energy for a given computation, but the total time required may be very different. Thus, other metrics are needed in order to include the notions of power and performance.

One such common measure that incorporates the speed of computation is performance per watt, often MIPS/W or million instructions per second per watt. This incorporates the power of the system into the energy efficiency metric. However, this measure is more appropriate for laptops or mobile devices, where reducing energy consumption is vital to save the battery. For many integrated circuit designs, power consumption is proportional to the square of the voltage, and reducing voltage by half reduces the frequency of the circuit by much less because of this quadratic term [19]. Thus one could easily sacrifice speed to get a better value for performance per watt. When speed of computation is a more vital factor, a different metric is warranted.

Gonzalez and Horowitz [9] propose a metric with a greater emphasis on performance, namely $\text{energy} \times \text{delay}$, in other words, the total energy used times the amount of time for the computation, with the lower the number the better. Brooks et al. [3] propose even squaring or cubing the delay, to give an even greater emphasis on performance.

We will use all three of these metrics to compare energy efficiency of the various implementations of the FDK reconstruction algorithm we examine in this paper.

Irregular Computed Tomography

CT reconstruction algorithms frequently consist of some variation of a back projection. The back projection operation in a CPU-based environment is typically coalesced with respect to the x-ray image or sinogram data. Neighboring voxels in the reconstruction volume will usually access image data from a small neighborhood of pixels of a given x-ray image. In a multi-threaded CPU-based implementation where the number of threads is on the order of 16 and each thread is updating its individually assigned voxel simultaneously, the memory access pattern can still be made relatively small (i.e. small enough so that CPU L1 and L2 cache hit-rates are still acceptable) by assigning neighboring voxels to CPU threads on the same multiprocessor. Regardless of the position of this neighborhood in the volume, the magnified projected neighborhood on the x-ray image is still reasonably small for almost any volume of interest and thus performance is not significantly degraded due to the memory access pattern.

In a GPU-based environment, hundreds to thousands of threads could potentially update

hundreds to thousands of voxels simultaneously. The magnified projection data footprint could be potentially enormous (at least larger than the on-chip cache structure can handle). Additionally, these thousands of threads do not run in lock-step; the consequence is neighboring threads potentially accessing entirely different x-ray images, or worse, updating entirely different image planes from a disjoint subset of images. In the CUDA environment, lock-step execution only occurs at the warp level (32 threads), with no control on warp execution ordering [5]. Therefore, the memory access pattern could easily become irregular over the large projection data neighborhood and throughout the entire reconstruction, thus severely hindering performance. This problem is exacerbated by the introduction of large-scale data; this include more projections, large magnifications, large images, large voxel counts, or any combination thereof. Although GPUs have worked very well on reconstruction algorithms on smaller scale datasets (medical datasets for example); for large-scale data, the algorithm would still require hours to days to reconstruct a 64 billion voxel volume.

Implementation

The implementation of this work revolves around the kernel design presented in the work by Jimenez et. al. [12] in order to achieve a majority of the energy metric improvement. The kernel design focuses on the general architecture of a graphics processor and exploits various features of the hardware such as:

1. *Fast Device Memory:* Data access can be faster and in parallel fetches on Graphics processors, this allows for faster evaluation of the bilinear interpolation step required of most back projection algorithms.
2. *Massive Multi-threading:* This is the most well known feature and appeal of GPGPU applications. For reconstruction, each thread is assigned a set of voxels to update, arranged such that no more than one voxel per image plane is assigned to a given thread.
3. *Texture Memory:* Fast read-only memory that has its own dedicated on-chip cache. Reading x-ray subimages through texture memory frees up L1-cache for voxel information and thus increasing computational (and ultimately energy) efficiency.
4. *Hardware Interpolation:* An additional benefit of utilizing texture memory is the exploitation of hardware-based interpolation. Although GPU-based interpolation is usually done in a lower precision, it has been shown to not affect numerical stability noticeably [17].
5. *Constant Memory:* To further reduce L1-cache pressure, all geometry parameters defining the arrangement of the CT system are stored in this user customizable on-chip cache.

The work by Jimenez et. al. also showed that in order to maximize voxel processing throughput by a GPU, one must mostly dedicate the device memory to voxel storage and upload small sets of x-ray projection data.

Implementing the irregular kernel is only half the task. Energy efficient optimizations made on the kernel will be moot if the host cannot provide the device with data fast enough in order to minimize device idling. To address the host side implementation, the method developed by Orr and Jimenez [20]. This approach attempts to minimize the GPU downtime by implementing an MIMD-like environment on the host.

Many traditional reconstruction algorithms, both CPU and GPU-based, execute tasks serially in a SIMD approach where a single thread on a CPU is tasked with storage I/O and compute/GPU kernel launch tasks. The consequence is that when a CPU thread is performing I/O tasks, the GPU is idling and not contributing towards the completion of any tasks while still consuming energy (albeit in a potentially lower power state).

The MIMD-like approach modularizes the read, launch and write tasks. Prior to reconstruction, threads are assigned a duty; either read/launch or read/write. Once assignments are made, the algorithm dynamically determines the amount of system memory available and calculates the number of image planes in the volume to reconstruct simultaneously; allowing for volatile memory storage of the relevant x-ray subimages necessary for the given subvolume and the subvolume itself. Next, all CPU threads perform the reading and pre-processing of the relevant x-ray data. Once reading has completed, a subset of threads (equal to the number of GPUs on the host) each determine its proportion of image planes to process at once, dependent on the GPU's hardware specifications and launches the required kernels while feeding necessary input data to the GPU. Once the reconstruction of the set of image planes is completed, it downloads the image planes to host memory and fetches the next set of image planes to reconstruct and repeats the process until all image planes of the subvolume are processed. The complement of host threads each iteratively check if any image planes are ready to be written. When the thread encounters an image plane that is ready to be written, it fetches the image plane, performs any required post-processing and writes the image plane to storage media. This process is repeated until all image planes of the subvolume are written to storage media. The entire process is repeated until the entire volume is reconstructed. The benefit of the approach is that the storage bottleneck is ameliorated by allowing writing tasks to occur during kernel computation while simultaneously reducing GPU downtime. The drawbacks to this approach is the additional load required on the CPUs while the write threads are iteratively checking for image planes to write and the additional memory required to temporarily store the image planes.

This work will present two implementations of irregular computed tomography; The first is a serialized approach based on the work of Jimenez et. al. and Jimenez and Orr [12, 11], the second is the modularized approach of Orr and Jimenez [20] which expands on the serialized approach by allowing overlapping compute and write tasks. Both implementations were written in C++ and Nvidia's CUDA programming environment (Ver. 5.0). For multiple GPU control, as well as the CPU thread assignment in the modularized approach, OpenMP 2.0 was utilized to implement CPU parallel tasking.

Evaluation

Three metrics will be measured:

1. *Energy Consumption*: This will be measured in kilowatt-hours (kWh).
2. *Performance-per-Watt*: Presented as average voxels reconstructed (and stored) per second per watt. Other more well-known performance-per-watt metrics, such as MFLOPs per watt, were not used as reconstruction is limited by the computation, the irregular memory access pattern, and the storage media I/O.
3. *Energy-Delay Product*: This metric is measured to ensure that the algorithm is not trading off energy savings for a slower reconstruction. In most applications, a delay in reconstruction is more detrimental than efficiency, we use a square weighting of delay as suggested by Laros III et. al. [16].

Metrics will be measured for four implementations of CT reconstruction. The first is a CPU-based multithreaded approach that uses both MPI and OpenMP to implement parallel processing. The CPU-based implementation is currently used in industrial radiography applications and will be used for comparison. Additionally, to serve as a GPU-based baseline, a naïve GPU-based approach is developed which consists of a brute force GPU-porting of CPU code which reconstructs one image plane per GPU iteratively and does not exploit the irregular nature of GPU-based reconstruction [12, 11]. The other two implementations are the serialized and modular approach described in the previous section.

The experiments were performed on a high-end Supermicro workstation that consists of dual octo-core Intel Xeon E-2687W processors clocked at 3.1 GHz with hyper-threading, 512 GB of system memory, 8 Nvidia Tesla M2090 GPUs ("Fermi"-class) in two Next I/O S2090 units connected via 4 PCI-E 2.0 x16 host interface cards, the storage media made up of 8 x 3GB SATA 6 Gb/s drives in a RAID 0 array controlled by an Intel Controller with 1 GB of DDR3 cache.

Energy metrics were obtained using several P3 International P4460 Kill-A-Watt EZ Electricity Usage Monitors directly connected to the workstation and each S2090 device. For the CPU-based implementation measurements, all GPUs were disconnected from the system before measurements are started to ensure GPU energy consumption was eliminated from the measurements.

Figure 5.1. Energy Consumption in kWh for the 64 gigavoxel reconstruction.

Results

Energy Consumption

Figure 5.1 shows energy consumption for all implementations when reconstructing the 64 gigavoxel dataset with respect to the number of GPUs. It is observed that the GPU-port actually consumes more energy than the CPU-based method when executed using 8 GPUs while both irregular approaches require approximately 1 kWh. For the GPU-port method, the GPUs are never fully utilized and as a result, a majority of the time is spent idling. Figure 5.2 shows energy consumption for the teravoxel dataset. For both irregular approaches, energy consumption improves with respect to GPUs as the reconstruction is completed in less time; it is noted that the improvement seems to level off towards 8 GPUs and is most likely due to PCI-E bus bandwidth limitations. Figures 5.1 and 5.1 show that overlapping compute and storage tasks does indeed reduce the energy consumption.

Figure 5.2. Energy Consumption in kWh for the teravoxel reconstruction. Note: vertical axis is presented in a log-scale

Performance Per Watt

Figures 5.3 and 5.4 show performance per watt with respect to the number of GPUs on the 64 gigavoxel and teravoxel datasets respectively. It is observed that while both irregular approaches significantly outperform the naïve and CPU-based approaches, the metrics do not improve with respect to GPUs, and in the serialized approach performance actually degrades. The leveled/degraded performance may be due to the fact that while 64 gigavoxels is indeed considered a large-scale dataset, the dataset may not properly push the limitations of the GPUs as is observed in the measurable improvement with respect to GPUs in the teravoxel dataset.

Energy-Delay Product

Figures 5.5 and 5.6 show the energy-delay product with respect to GPUs on the 64 gigavoxel and teravoxel dataset respectively. These figures validate that the irregular approaches do not trade off efficiency for delay. In fact, for the 64 gigavoxel dataset, the energy-delay product increases by over an order of magnitude for the non-irregular GPU-based approach. Figure 5.5 shows that while the energy-delay product for the modular approach slightly increases for 8 GPUs, it is still an improvement over all other approaches.

Figure 5.3. Performance per watt for the 64 gigavoxel reconstruction

For the teravoxel dataset, the reconstruction pushes all approaches to their limits and thus we see a continued improvement in the irregular approaches with respect to GPU count. In both methods, we see that both irregular approaches achieve an improved energy-delay product by 3 orders of magnitude compared to the CPU-based approach.

Conclusion

It has been well documented that computed tomography reconstruction algorithms can greatly benefit from the utilization of GPGPU technology. This has been observed by the remarkable increase in computational performance and reduction in time required to execute the reconstruction task. We have shown that GPU-based approaches also benefit from energy efficiency performance; not just in overall energy consumption, but other energy efficiency metrics as well. As the computing community approaches energy limitations, intelligent algorithm design will be crucial for the exploitation of optimal performance from the hardware. The community needs to explore other approaches outside of reconstruction and investigate whether some algorithms can be further improved by implementing energy efficient methods.

Figure 5.4. Performance per watt for the teravoxel reconstruction

Figure 5.5. Energy-Delay Product for the 64 gigavoxel reconstruction

Figure 5.6. Energy-Delay Product for the teravoxel reconstruction

References

- [1] C Axelsson and P Danielsson. Three-dimensional reconstruction from cone-beam data in $O(n^3 \log n)$ time. *Physics in Medicine and Biology*, 39(3):477, 1994.
- [2] Harrison H. Barrett and Kyle J. Myers. *Foundations of Image Science*. Wiley-Interscience, 2004.
- [3] D.M. Brooks, P. Bose, S.E. Schuster, H. Jacobson, P.N. Kudva, A. Buyuktosunoglu, J.-D. Wellman, V. Zyuban, M. Gupta, and P.W. Cook. Power-aware microarchitecture: design and modeling challenges for next-generation microprocessors. *Micro, IEEE*, 20(6):26–44, 2000.
- [4] Nvidia Corporation. *CUDA C Programming Guide v5.0*. <http://www.nvidia.com>, 2012.
- [5] Nvidia Corporation. *Nvidia CUDA C Best Practices Guide v5.0*. <http://www.nvidia.com>, 2012.
- [6] O.S. Cossairt, D. Miao, and S.K. Nayar. Gigapixel computational imaging. In *Computational Photography (ICCP), 2011 IEEE International Conference on*, pages 1–8, 2011.
- [7] Nibir K. Dhar and Ravi Dat. Advanced imaging research and development at darpa. In *Proceedings of SPIE, Infrared Technology and Applications XXXVIII*, volume 8353. SPIE, 2012.
- [8] L.A. Feldkamp, L.C. Davis, and J.W. Kress. Practical cone-beam algorithm. *Journal of the Optical Society of America A*, 1(6):612–619, 1984.
- [9] R. Gonzalez and M. Horowitz. Energy dissipation in general purpose microprocessors. *Solid-State Circuits, IEEE Journal of*, 31(9):1277–1284, 1996.
- [10] S. Izumi, S. Kamata, K. Satoh, and H. Miyai. High energy x-ray computed tomography for industrial applications. *Nuclear Science, IEEE Transactions on*, 40(2):158–161, apr 1993.
- [11] Edward S. Jimenez and Laurel J. Orr. Rethinking the Union of Computed Tomography Reconstruction and GPGPU Computing. In *Workshop on Penetrating Radiation Systems and Applications XIV*, SPIE Optical Engineering + Applications, August 2013.
- [12] Edward S. Jimenez, Laurel J. Orr, and Kyle R. Thompson. An Irregular Approach to Large-Scale Computed Tomography on Multiple Graphics Processors Improves Voxel Processing Throughput. In *Workshop on Irregular Applications: Architectures and Algorithms*, The International Conference for High Performance Computing, Networking, Storage and Analysis, November 2012.

- [13] Edward S. Jimenez, Laurel J. Orr, and Kyle R. Thompson. An irregular approach to large-scale computed tomography on multiple graphics processors improves voxel processing throughput. In *Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, pages 254–260. IEEE Computer Society, 2012.
- [14] Ralf Kohley, Philippe Gar, Cyril Vtel, Denis Marchais, and Franois Chassat. Gaia’s fpa: Sampling the sky in silicon. In *Proceedings of SPIE, Space Telescopes and Instrumentation: Optical, Infrared, and Millimeter Wave*, volume 8442. SPIE, 2012.
- [15] Monica D. Lam, Edward E. Rothberg, and Michael 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, ASPLOS-IV*, pages 63–74, New York, NY, USA, 1991. ACM.
- [16] James H. Laros III, Kevin Pedretti, SuzanneM. Kelly, Wei Shu, Kurt Ferreira, John Vandyke, and Courtenay Vaughan. In *Energy-Efficient High Performance Computing*, SpringerBriefs in Computer Science. 2013.
- [17] Wen mei W. Hwu, editor. *GPU Computing Gems - Emerald Edition*. Morgan Kaufmann, 2011.
- [18] Todd Mowry and Anoop Gupta. Tolerating latency through software-controlled prefetching in shared-memory multiprocessors. *Journal of Parallel and Distributed Computing*, 12(2):87 – 106, 1991.
- [19] T. Mudge. Power: a first-class architectural design constraint. *Computer*, 34(4):52–58, 2001.
- [20] Laurel J. Orr and Edward S. Jimenez. Preparing for the 100-megapixel Detector: Reconstructing a Multi-Terabyte Computed-Tomography Dataset. In *Workshop on Penetrating Radiation Systems and Applications XIV*, SPIE Optical Engineering + Applications, August 2013.
- [21] Eric Pепенhausen, Ziyi Zheng, and Klaus Mueller. Gpu-accelerated back-projection revisited: Squeezing performance by careful tuning. In *Workshop on High Performance Image Reconstruction*, Postdam, Germany, 2011.
- [22] Jason Sanders and Edward Kandrot. *CUDA By Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley, 2011.
- [23] S. Xiao, Y. Bresler, and Jr. Munson, D.C. Fast feldkamp algorithm for cone-beam computer tomography. In *Image Processing, 2003. ICIP 2003. Proceedings. 2003 International Conference on*, volume 2, pages II – 819–22 vol.3, sept. 2003.
- [24] F. Xu and K. Mueller. Accelerating popular tomographic reconstruction algorithms on commodity pc graphics hardware. *Nuclear Science, IEEE Transactions on*, 52(3):654 – 663, june 2005.

- [25] Fang Xu and K. Mueller. Ultra-fast 3d filtered backprojection on commodity graphics hardware. In *Biomedical Imaging: Nano to Macro, 2004. IEEE International Symposium on*, pages 571 – 574 Vol. 1, april 2004.
- [26] Fang Xu and Klaus Mueller. Real-time 3d computed tomographic reconstruction using commodity graphics hardware. *Physics in Medicine and Biology*, 52(12):3405–3419, 2007.

Appendix A

Technology Advances

1. *A High-Performance Multi-GPU based Large Scale Radiography and Computed Tomography Dataset Simulator*. Edward S. Jimenez, Ismael Perez and Kyle R. Thompson. August 2013 (in review)
2. *Cluster-Based Modularized Approach to GPU-based Computed Tomography with Non-Linear Load Balancing and Node-Interconnect Optimizations*. Edward S. Jimenez and Laurel J. Orr. July 2013 (in review)
3. *Modularized Approach to GPU-Based Computed Tomography Improves Global Performance in High-Performance Systems*. Edward S. Jimenez and Laurel J. Orr, February 2013 (provisional patent filed July 2013)
4. *Irregular Approach to Large-Scale Computed Tomography on GPUs*. Edward Jimenez, September 11 2012 (in review)

Appendix B

Publications and Presentations

1. *A High-Performance and Energy-Efficient CT Reconstruction Algorithm to Reconstruct Multi-Terabyte Datasets.* Edward S. Jimenez, Laurel J. Orr, and Kyle R. Thompson. **Conference Proceedings for the IEEE Nuclear Science Symposium and Medical Imaging Conference, NSS-MIC 2013.** October 2013 Seoul, Korea.
2. *Irregular Large-Scale Computed Tomography on Multiple Graphics Processors Improves Multiple Energy-Efficiency Metrics for Industrial Applications.* Edward S. Jimenez, Laurel J. Orr, and Kyle R. Thompson. **Conference Proceedings for the Conference on High Performance Computing Networking, Storage and Analysis, SC 2013, Workshop on Irregular Applications: Architectures and Algorithms.** November 2013. Denver CO.
3. *High Performance Cluster-based Approach to GPU-based Computed Tomography.* Laurel J. Orr, Edward S. Jimenez, and Kyle R. Thompson. **Conference on High Performance Computing Networking, Storage and Analysis, SC 2013.** November 2013. Denver CO
4. *Rethinking the Union of Computed Tomography Reconstruction and GPGPU Computing for Industrial Applications.* Edward S. Jimenez and Laurel J. Orr. **Conference Proceedings for the Penetrating Radiation Systems and Applications XIV workshop at the SPIE International Symposium on SPIE Optical Engineering+Applications.** San Diego CA August 25-29 2013.
5. *Preparing for the 100-Megapixel Detector: Reconstructing a Multi-Terabyte Computed Tomography Dataset.* Laurel J. Orr and Edward S. Jimenez. **Conference Proceedings for the Penetrating Radiation Systems and Applications XIV workshop at the SPIE International Symposium on SPIE Optical Engineering+Applications.** San Diego CA August 25-29 2013.
6. *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. **Conference Proceedings for the Conference on High Performance Computing Networking, Storage and Analysis, SC 2012, Workshop on Irregular Applications: Architectures and Algorithms.** Salt Lake City, UT, USA, November 11 2012.

7. *Improving the Quality of Life Through Video Games*. Edward S. Jimenez. **Internal Sandia Seminar for Org. 9515**. March 2012.
8. *An Innovative Irregular Approach to Computed Tomography Reconstruction Using Graphics Processors*. Edward S. Jimenez. **Labs Accomplishments February 2013, Sandia Lab News**. Page 9.
9. *High Performance Graphics Processor-based Computed Tomography Reconstruction Algorithms*. Edward S. Jimenez, **Poster presentation at LDRD Day**, held in Albuquerque NM on August 13, 2013.
10. *Rethinking the Union of Computed Tomography Reconstruction and GPGPU Computing for Industrial Applications*. Edward S. Jimenez, invited talk at the **San Diego State University Computation Science Research Center Spring Colloquium Series**. San Diego CA. March 15, 2013.
11. *High Performance Graphics Processor based Computed Tomography Reconstruction Algorithms*. Edward S. Jimenez, Oral Presentation at Sandia National Laboratories **DataCon 2012**, August 23, 2012.
12. *Optimized HPC Approach to CT Reconstruction with GPUs*. Laurel J. Orr, Carson Kent, and Edward S. Jimenez, Poster Presentation at Sandia National Laboratories **DataCon 2012**. August 22, 2012.

DISTRIBUTION:

1	MS 0531	Michael O. Vahle, 9000
1	MS 0557	David Epp, 1522
1	MS 0932	Judy E. Spomer, 9515
1	MS 0933	Joselyne O. Gallegos, 9500
1	MS 0933	Barry V. Hess, 9510
1	MS 0899	Technical Library, 9536 (electronic copy)
1	MS 0161	Legal Intellectual Property, 11500
1	MS 0359	D. Chavez, LDRD Office, 1911