

MULTI GPU IMPLEMENTATION OF ITERATIVE TOMOGRAPHIC RECONSTRUCTION ALGORITHMS

Byunghyun Jang, David Kaeli

Northeastern University
Department of ECE
Boston, MA U.S.A.

Synho Do, Homer Pien

Massachusetts General Hospital
Department of Radiology
Boston, MA U.S.A.

ABSTRACT

Although iterative reconstruction techniques (IRTs) have been shown to produce images of superior quality over conventional filtered back projection (FBP) based algorithms, the use of IRT in a clinical setting has been hampered by the significant computational demands of these algorithms. In this paper we present results of our efforts to overcome this hurdle by exploiting the combined computational power of multiple graphical processing units (GPUs). We have implemented forward and backward projection steps of reconstruction on an NVIDIA Tesla S870 hardware using CUDA. We have been able to accelerate forward projection by 71x and backward projection by 137x. We generate these results with no perceptible difference in image quality between the GPU and serial CPU implementations. This work illustrates the power of using commercial off-the-shelf low-cost GPUs, potentially allowing IRT tomographic image reconstruction to be run in near real time, lowering the barrier to entry of IRT, and enabling deployment in the clinic.

Index Terms— Iterative reconstruction, GPU, Computed tomography

1. INTRODUCTION

Advances in Computed Tomography (CT) technology and associated reconstruction algorithms have increased the need for increased computing performance. In particular, iterative reconstruction techniques (IRTs) are computationally demanding and have not been adopted in clinical settings due to their computational requirements, despite their superiority over filtered back projections (FBP). Previous work has looked at how to begin to address these problems [1, 2]

Because of recent advances in computer hardware architecture, such as multi-core CPUs and general purpose GPUs, a variety of research fields have started to retool their applications to harness this unprecedented rate of growth in computing power. GPUs, in particular, have been receiving special attention because of their tremendous GFLOP/dollar,

GFLOP/watt, and their general availability on desktop PCs. Programming difficulties on General Purpose GPUs (GPGPUs) using traditional graphic APIs have been recently resolved with the introduction of the CUDA programming environment. CUDA makes it possible to run general purpose parallel programs fairly easily, taking advantage of the GPU's tremendous computing power. Driven by these advances in high performance computing, GPUs are becoming the platform of choice for data-parallel compute-intensive applications in a number of fields.

In this paper, we explore how best to map IRT algorithms to a system containing multiple GPUs. We report on the resulting speedup of these algorithms, and discuss specific optimizations that we apply. Our results show that using the raw data acquired from a Siemens Sensation-64, our GPU-based IRT implementation achieved a 71x speedup for forward projection and a speedup of 137x for back projection (compared to a serial implementation on high end CPU), without any modification of the original algorithm and without any loss of image quality. Our work demonstrates that IRT can potentially be deployed in a clinical setting. To the best of our knowledge, this work is the first attempt to utilize multiple GPUs for tomographic image reconstruction. We report on how we obtained such large speedups, and issues remaining in this work.

2. BACKGROUND

2.1. Tesla S870 and CUDA

The Tesla series is the first GPU that is dedicated to general purpose computing. Tesla was introduced by NVIDIA in 2007 to enable desktop supercomputing. The NVIDIA Tesla S870 (server version) integrates four Tesla C870 GPUs; the C870 is based on NVIDIA's G80 microarchitecture. The G80 microarchitecture consists of 16 streaming multiprocessors (SMs), each containing 8 streaming cores running at 1.35 GHz. Each SM has 8,192 registers and 16 KB shared memory that are shared among all threads assigned to the SM. Below are some highlights of the S870 hardware.

To allow users to more easily interface to GPUs, NVIDIA

Thanks to NVIDIA for generous hardware donations

# of Streaming Processor Cores	512
Frequency of Processor Core	1.35 GHz
Floating Point Precision	IEEE 754 Single
Total Dedicated Memory	6 GB GDDR3
Memory Speed	800 MHz
Memory Bandwidth	76.8 GB/s
Peak Performance	2 TFLOPs

Table 1. Hardware Specification of the Tesla S870.

introduced a software framework called CUDA, which is an extension of the standard C/C++ languages. The programming model of CUDA is *single instruction multiple threads (SIMT)* where a single instruction is executed in multiple threads. Each thread specified by the programmer is mapped to one scalar processor core by the SIMT unit of the stream multiprocessor. Each thread is able to be executed independently, with its own instruction address and register state, and a group of 32 threads called warps are created, managed, and scheduled by the SIMT unit. Consequently, hundreds or thousands of threads are executed simultaneously, providing the ability to hide many memory latencies incurred on the GPU.

While many programmers now enjoy the benefits of CUDA, there is a general lack of support for optimizing the resulting CUDA code. Optimization would require a deeper understanding of the underlying hardware. Many of the recent work on GPUs has focused on optimizations [3, 4, 5]. We present some of key optimization techniques used in this work in the following sections.

2.2. Iterative Reconstruction Techniques (IRTs)

Tomographic reconstruction is pervasive in medical imaging. X-ray transmission computed tomography (CT) is conventionally solved analytically, by applying filtered back projections (FBP) based approaches. While FBP has helped to bring about considerable advances in medical imaging, the current push for larger coverage cone-beam CT systems, at lower radiation doses with higher resolution, have motivated researchers to consider alternative image reconstruction schemes. Because IRT iteratively estimates the image using a model of the scanner’s imaging geometry, IRTs can produce images that are more robust to noise and artifacts, and are of higher resolution and quality overall [6].

The penalty for using IRT, however, is computational overhead. By making small corrections at each iteration, IRT is computationally intensive, requiring either numerous CPUs working in concert, or long delays in image formation. As such, GPUs offer an attractive alternative.

In this paper, a least squares reconstruction formulation is used. Let f denote the image to be reconstructed, g be the observed projection data, and H the projection operator which maps f into g . Then the least-squares solution is given

by the minimization of $E = \|g - Hf\|^2$. Although more sophisticated formulations with different prior models can be imposed, the principles of using GPUs to compute forward and back projections can best be illustrated using a simple formulation. Note in particular that, as the area of coverage gets larger in modern multi-detector CT systems, FBP requires approximations to interpolate cone-beams into parallel fan-beams, and fan-beams into parallel lines of projection, the iterative framework requires no such approximation. That is, as long as the H captures the imaging geometry with high fidelity, no such approximations are necessary.

3. GPU IMPLEMENTATIONS

3.1. Implementation and Optimizations on single GPU

The first step toward a GPU implementation is to determine which portion of the application should be run on the GPU. Generally, in data parallel computation, nested loops provide for the greatest amount of explicit data parallelism. Thus, the forward projection and back projection building blocks of IRT are chosen to be offloaded. In cases where the depth of nesting is deep (e.g. the forward projection routine), the programmer must select which inner loop to be offloaded onto the GPU. There are trade-offs between utilizing large and small kernels. Large kernels are challenging to optimize due to the large range of optimization variables and unpredictable interplay between a large number of threads. Targeting smaller kernels can be ineffective due to insufficient arithmetic intensity (i.e., the ratio between ALU and memory instructions) which is necessary to benefit from massive multithreading.

If a kernel has multiple nested loop, the next step is to determine which loop to map to a thread. Since the GPU requires a sufficiently large number of threads to not only hide memory latency, but to compensate for the overhead associated with hardware pipelining, loops with the the largest trip counts (i.e., iterations) is generally the best choice (each iteration of loop becomes a thread). We chose to generate threads on an angle basis for forward projection and on a voxel basis in back projection. We found that we could not use angles to generate threads in back projection due to a case where multiple threads in the same warp try to access same memory location and this can lead to nondeterministic results.

The first optimization to pursue in CUDA programming is to tune two level thread hierarchy called *execution configuration*. A execution configuration is defined by the thread block (a group of threads) and thread grid (a group of thread blocks). Selecting the correct size for a thread block is particularly key for performance since it determines the number of threads that can be run simultaneously. CUDA provides a handy tool called the *occupancy calculator* which allows the programmer to easily calculate the best thread block size based on register and shared memory usage of a kernel. A

number of previous work has proposed efficient methods related to this optimization space.

The most effective and important optimization opportunities are to explore efficient utilization of the GPU memory units. The peak performance of the Tesla S870 is 2 TFLOPs, although this is only achievable if we can hide all memory latencies. There are different types of memories on the G80 series hardware; our optimizations need to consider how best to utilize this space for the given application. In our implementation, the read-only data possessing 2D locality is mapped to texture memory, while the read-only data that exhibits 1D locality is stored in constant memory. The fastest shared memory is used for temporary storage whenever possible. The effect of this is reflected in 3rd bar in figure 3 labeled *Memory*.

Another effective optimization is to utilize caching. Texture and constant memory on the GPU uses caching and so we maximize the preservation of spatial and temporal data locality to help improve performance. To increase cache performance in our implementation, the input data that is mapped to constant memory is loaded as one big chunk based on the program's access pattern. The effect of this optimization is reflected in 4th bar in figure 3 labeled as *Caching*.

Other optimization techniques considered include register usage minimization and utilization of the device runtime mathematical functions. Note that registers are the hardware resource that directly affects the number of active threads. The device runtime mathematical functions run in less time, but produce less accurate results, so they should only be used where precision is not critical. These techniques are included in our implementation, even though their impact on performance is not shown explicitly in the graph.

3.2. Multi GPUs

The NVIDIA Tesla S870 is equipped with four independent C870 GPUs. Therefore, the programmer must decide how many GPUs will be used and how each GPU will be exploited. Since we aim at improving kernel execution time rather than utilizing each GPU differently, the first thing to do is to determine a workload distribution. There are two schemes: one is to distribute the number of threads to be executed among GPUs and the other is to divide workloads inside the kernel body. In either case, we need to merge outputs from each GPU to get final results.

To minimize the overhead that occurs in data copying, kernel invocation, etc., we create the same number of CPU threads as GPUs to be utilized, each of which takes care of an individual GPU. Each thread copies input data from the CPU to the GPU, executes the kernel, and copies results back to the CPU. The host CPU waits for all CPU threads to complete and merges results into one. This process is illustrated in figure 1.

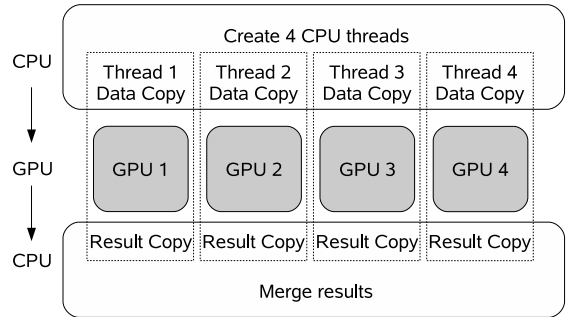


Fig. 1. Multi GPU Utilizations

4. RESULTS AND CONCLUSIONS

To compare both performance and image quality, we used as a baseline a standard PC with a Intel Core 2 Duo processor clocked at 2.66 GHz, with 4 MB cache and 2 GB of memory. Raw data of a cadaveric heart image obtained from a Siemens Sensation-64 MDCT scanner is used. The image field of view of $350 \times 350 \times 9$ is reconstructed from data obtained from 1280 angles. Figure 2 shows the resulting image reconstruction from running 30 iterations on both GPU and CPU. Figure 2(c) shows the difference between GPU and CPU implementations - note that the mean squared difference is approximately 5×10^{-4} , a difference that is not perceptible.

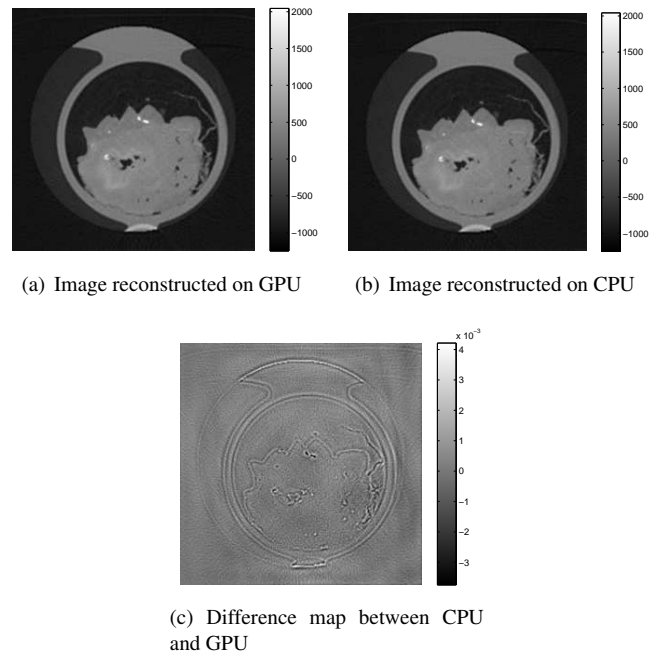


Fig. 2. Comparison of images reconstructed on the GPU and a CPU.

Figures 3 and 4 show the performance results and decomposition of the execution time, respectively. The leftmost bar

in figure 3 is the execution of our baseline serial version and the rest of the bars show results run on a GPU with the specific optimizations discussed in section 3. Note that in the figure, we just continue to add each optimization as we move from left to right. We clearly see that optimizations are effective and our multi-GPU approach is beneficial for both cases. However, we found that our multi-GPU implementation encounters significant overhead due to CPU thread synchronization. This is caused by the operating system's thread scheduling policy, and its impact can be seen in back projection in figure 4. This overhead may cause serious problem in the case of small kernels whose execution time is very short. If this problem can be avoided, we estimate that a 234x speedup in back projection may be achievable.

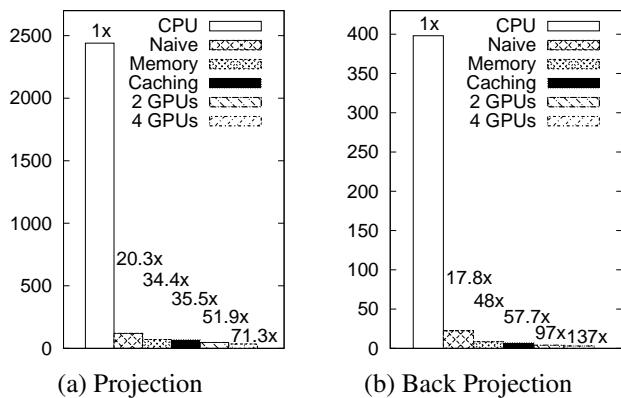


Fig. 3. Performance Comparison (Numbers in y-axis is in seconds and numbers on top of each bar indicate speedup over a CPU implementation. Note also that 2nd, 3rd, and 4th bar are the results of running on a single GPU.)

We found that a GPU is a great choice of platform for tomographic reconstruction in terms of performance, cost, and availability, enabling many exciting possibilities in a clinical setting. Image quality (which was a big concern in earlier GPU implementations [7]) is no longer a concern with the current generation of GPUs that supports single precision floating point as a default. For precision-critical applications, the latest NVIDIA GPUs are capable of double precision floating point, though with some performance degradation.

5. REFERENCES

[1] Klaus Mueller, Fang Xu, and Neophytos Neophytou, "Why do commodity graphics hardware boards (GPUs) work so well for acceleration of computed tomography?," 2007, vol. 6498, p. 64980N, SPIE.

[2] K. Mueller and R. Yagel, "Rapid 3-D cone-beam reconstruction with the simultaneous algebraic reconstruction technique (SART) using 2-D texture mapping hardware,"

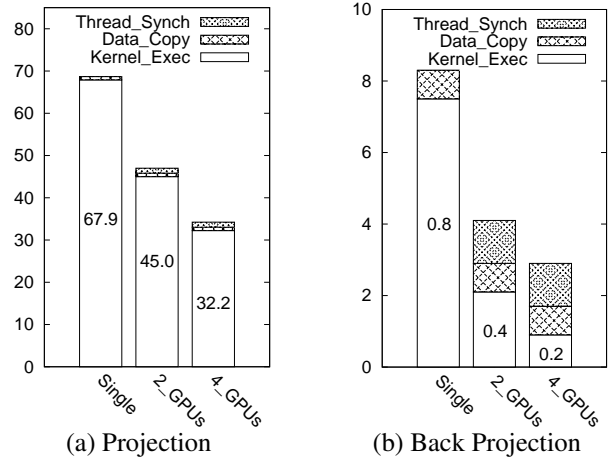


Fig. 4. Decomposition of Execution Time (All numbers are in seconds. Note that there is no thread synchronization component on single GPU implementation)

Medical Imaging, IEEE Transactions on, vol. 19, no. 12, pp. 1227–1237, Dec. 2000.

[3] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "GPU Computing," in *Proceedings of the IEEE*, 2008, vol. 96, pp. 879–899.

[4] Mark Silberstein, Assaf Schuster, Dan Geiger, Anjul Patney, and John D. Owens, "Efficient computation of sum-products on gpus through software-managed cache," in *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*, New York, NY, USA, 2008, pp. 309–318, ACM.

[5] Samuel S. Stone, Justin P. Haldar, Stephanie C. Tsao, Wen mei W. Hwu, Zhi-Pei Liang, and Bradley P. Sutton, "Accelerating advanced MRI reconstructions on gpus," in *CF '08: Proceedings of the 2008 conference on Computing frontiers*, New York, NY, USA, 2008, pp. 261–272, ACM.

[6] Jean-Baptiste Thibault, Ken D. Sauer, Charles A. Bouman, and Jiang Hsieh, "A three-dimensional statistical approach to improved image quality for multislice helical CT," *Medical Physics*, vol. 34, no. 11, pp. 4526–4544, 2007.

[7] K. Mueller and Fang Xu, "Practical considerations for GPU-accelerated CT," *Biomedical Imaging: Nano to Macro, 2006. 3rd IEEE International Symposium on*, pp. 1184–1187, April 2006.