GPU-Accelerated Back-Projection Revisited: Squeezing Performance by Careful Tuning

Eric Papenhausen, Ziyi Zheng and Klaus Mueller

Abstract— **In recent years, GPUs have become an increasingly popular tool in computed tomography (CT) reconstruction. In this paper, we discuss performance optimization techniques for a GPU-based filtered-backprojection reconstruction implementation. We explore the different optimization techniques we used and explain how those techniques affected performance. Our results show a nearly 50% increase in performance when compared to the current top ranked GPU implementation.**

Index Terms—**GPU, FDK, CUDA, Computed Tomography**

I. Introduction

GPU-accelerated CT reconstruction has been demonstrated as a fast and practical solution for medical and industrial CT. One of the most popular reconstruction methods is the FDK algorithm [1] which can provide high-resolution reconstruction results. The FDK algorithm consists of two stages, filtering and backprojection, in which backprojection is the most time consuming part. There has been widespread coverage of research on high performance backprojection algorithms [2][3][4][5][6][7]. However, until recently, there has not been an effective way of comparing CT reconstruction implementation performance. Thanks to the great efforts made by Rohkohl et al. [8] who created the RabbitCT platform [9] as a standardized framework of comparing CT implementations, such undertakings can now be easily accomplished. Using RabbitCT, it is possible to evaluate the performance of different CT backprojection implementations. In this paper, we focus our efforts on GPU-optimization and we take advantage of the RabbitCT platform to benchmark the performance. The experimental results show that our fully optimized GPU implementation has the fastest speed comparable to other implementation's results shown in the website ranking.

In this paper, we begin in Section II by discussing the related work and giving a brief description of the GPU architecture. In Section III, we describe how we approached the problem. In Sections IV, V, and VI we present three major configurations that we implemented. Section VII shows some of the optimization techniques tried. Section VIII presents results and Section IX concludes our paper.

II. Related Work and Background

The first attempt to use graphics hardware for CT reconstruction has been by Cabral et al [10] using filtered backprojection. Mueller and Yagel implemented [11] iterative reconstruction algorithms on graphics hardware. Both of these two works were based on the texture mapping suite resident in non-programmable high-end SGI workstations. Later with the development of a new generation of programmable GPUs, Xu and Mueller [12][13] demonstrated GPU-accelerated CT reconstruction and had vast speedups, with respect to their CPU counterparts. With the recent development of GPU architectures into main stream multiprocessors, the GPU has unleashed its computational power such that it is no longer restricted to graphics applications (or applications that have to mimic graphics applications), extending their use to general purpose applications. Scherl et al [2] implemented CT reconstruction with the new general purpose computing API, CUDA. CUDA API helped researchers to notice that CT reconstruction on GPU is a memory-bounded rather than computational-bounded problem. Square-type [4] and line-type [5] decomposition have been explored to optimize memory bandwith. Zheng and Mueller [3] addressed the optimization of cache usage. Jia et al [7] implement iterative reconstruction method based on solving minimization problem in CUDA.

Modern GPUs follow the "Single Instruction, Multiple Thread" (SIMT) model of parallel execution. This method of execution is ideal for the back-projection algorithm we implement. A C-like API called CUDA (Computer Unified Device Architecture) is used to program the NVIDIA GPUs.

The GPU used in our experiments was the NVIDIA GeForce GTX 480. This graphics card has 15 streaming multiprocessors; each containing 32 cores. Its theoretical computing power is approximately 1.3 TFLOPS. This GPU, like all modern GPUs, has off-chip memory and on-chip caching mechanisms. Off-chip memory, also known as device memory, incurs hundreds of cycles of memory latency and is often the bottleneck of a GPU accelerated application. Off-chip memory includes global, texture, and constant memory. However, texture and constant memory can be cached, replacing the hundreds of cycles of latency with only a few cycles of latency for the on-chip cache. The GTX 480 has a peak memory bandwidth of 177.4 GB/s for its 1.5 GB DDR5 device memory.

It was important that any access to global memory be performed in a coalesced fashion. A coalesced memory access implies that multiple memory locations are returned in a single access. A coalesced memory access will occur when all the threads in a warp access consecutive memory locations.

Execution of a task by a CUDA kernel is organized into thread blocks. Thread blocks are organized into a grid. The GTX 480 can have a maximum of 1024 threads / thread block.

Eric Papenhausen, Ziyi Zheng and Klaus Mueller are with the Computer Science Department, Stony brook University, Stony Brook, NY 11777 USA (phone: 631-632-1524; e-mail: {papenhausen, zizhen, mueller}@ cs.sunysb.edu).

III. Overall Methodology

Our implementation is loosely based on the work performed in [4]. We use a voxel parallelism approach in which each thread block would calculate a section of the three dimensional image. More specifically, each thread computes an array of voxels in the Z direction as shown in figure 1. After experimenting with different thread block dimensions, we found that a dimension of $16 \times 16 \times 4$ yielded the best performance.

We quickly realized, however, that memory bandwidth was the bottleneck. We turned our focus to reducing the total number of global memory accesses. Access to global memory can take 400 to 600 clock cycles of memory latency. In the naïve GPU implementation, we have 15 global memory accesses per kernel invocation. Our fully optimized configuration, however, only has two global memory accesses per kernel invocation. We experimented with un-coalesced memory accesses and found that it reduced performance by up to 60 percent.

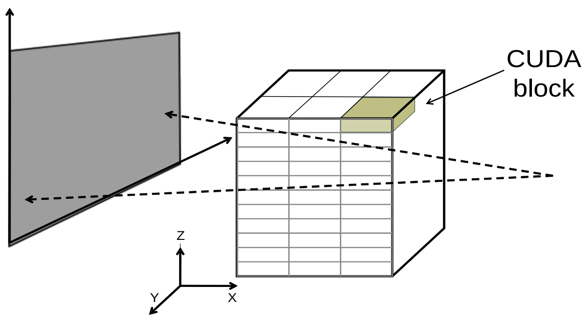

Figure 1. Voxel-driven method in back-projection.

IV. Naïve Configuration

The RabbitCT website [9] provided us with a simple CPU implementation of the backprojection algorithm and a dataset containing 496 projections. We decided to develop a naïve GPU based implementation so we could see the effects of further optimizations. For each kernel invocation, this configuration involved storing both the projection image **I**, and its corresponding projection matrix **A**, in global memory.

Figure 2 shows pseudo code for this method. This implementation has many global memory accesses. There are approximately four floating-point operations per memory access. Increasing the number of operations per memory access is critical in memory intensive applications such is this. In this configuration, we do not fully use all the capabilities the GPU has to offer. We explicitly wrote code to perform bilinear interpolation. The GPU has a hardware mechanism dedicated to performing interpolation. By performing interpolation explicitly, we increase the number of registers needed by the kernel. This led to a decrease in occupancy.

Occupancy is very important when it comes to performance. The number of threads that can run on a streaming multiprocessor determines occupancy. Occupancy is critical in latency hiding. A high occupancy indicates that many warps fit into a single streaming multiprocessor. This allows the GPU to hide latency incurred by memory accesses of one warp, by allowing another warp to execute. Occupancy has a number of limiting factors. In our experience, the most common limiting factor is register usage. Although a high occupancy does not always lead to an increase in performance, its role in memory intensive programs is critical.

```
row = blockIdx.y * blockDim.y + threadIdx.y
col = blockIdx.x * blockDim.x + threadIdx.x
FOR k = 0 to L
 x = O_L +col * R_L
 y = O_L + row * R_L
 z = O_L + k * R_L

 w = A[2] * x + A[5] * y + A[8] * z + A[11]
 u = (A[0] * x + A[3] * y + A[6] * z +A[9]) / w
 v = (A[1] * x + A[4] * y + A[7] * z + A[10]) / w

 result = interpolate (u, v)
 result = result / $w^2$
 f_L[k * $L^2$ + row * L + col] += result
END
```

Figure 2: Pseudo-code for the naïve configuration. The call to “interpolate” is the implemented interpolation algorithm from [8].

V. Application Specific Integrated Circuits

This configuration addresses some of the issues presented in Section IV. Our implementation utilizes Application Specific Integrated Circuits (ASICs) on the GPU for fast 2D texture interpolation. This is done through the use of texture memory and constant memory. ASICs are special circuits that are designed for a specific task. In this configuration, we store the projection image **I**, in texture memory. Texture memory is cached, and allows us to take advantage of the GPU’s hardware interpolation mechanism.

Figure 3 shows the pseudo code for this implementation. We can see that the interpolation method from figure 1 has been replaced with a call to the tex2D method. This method performs bilinear interpolation for us.

```
texture<float, 2> texRef
__constant__ float A[12]

row = blockIdx.y * blockDim.y + threadIdx.y
col = blockIdx.x * blockDim.x + threadIdx.x
FOR k = 0 to L
      result = f_L[k * $L^2$ + row * L + col]

      x = O_L +col * R_L
      y = O_L + row * R_L
      z = O_L + k * R_L

      w = A[2] * x + A[5] * y + A[8] * z + A[11]
      u = (A[0] * x + A[3] * y + A[6] * z +A[9]) / w
      v = (A[1] * x + A[4] * y + A[7] * z + A[10]) / w

     result += tex2D ( texRef, (u + 0.5), (v + 0.5)) / $w^2$
     f_L[k * $L^2$ + row * L + col] = result
END
```

Figure 3: Pseudocode for the ASIC implementation

The use of texture memory and its interpolation mechanism gave us a large performance increase for a few reasons. First, the hardware based bilinear interpolation is faster than the user implemented interpolation from Section IV. Second, this allowed us to free up registers, which led to an increase in occupancy. Finally, since texture memory is cached, memory latency was only incurred on a cache miss. It is also important to note that the CUDA based interpolation described in [14], is slightly different from the method of interpolation of [8]. This is why we add the coordinates in the tex2D method by 0.5.

We store the projection matrix **A** in constant memory. Constant memory is also cached and works best when each thread is accessing the same memory location. The use of constant memory yielded a significant performance increase; even before texture memory was used. The use of constant and texture memory allowed us to reduce the total number of global memory accesses to two per kernel invocation. This is, however, a lower bound because each cache miss will incur a global memory access.

VI. Multiple Projections

While examining our implementation from Section V, we realized that for each kernel invocation, a given thread operated on the same array of voxels. The only difference was in the data provided by the projection image **I**, and the projection matrix **A**. With this information, we saw an opportunity to further increase performance by letting the GPU operate on multiple projections for each kernel invocation.

After some experimentation, we found that passing 4 projections per kernel invocation yielded the best performance. The benefits are that this reduces the total number of global memory accesses as well as the total number of kernel invocations by a factor of 4. The pseudo code in Figure 4 shows that there are still two global memory accesses per kernel invocation; however, each memory access is being used more effectively because we are operating on 4 projections instead of 1.

```
texture<float, 2> tRef, tRef2, tRef3, tRef4
__constant__ float A[48]

row = blockIdx.y * blockDim.y + threadIdx.y
col = blockIdx.x * blockDim.x + threadIdx.x
FOR k = 0 to L
    result = f_L[k * $L^2$ + row * L + col]

    x = O_L +col * R_L
    y = O_L + row * R_L
    z = O_L + k * R_L

    // mapping voxel (x,y,z) to projection 1 and backproject

    w = A[2] * x + A[5] * y + A[8] * z + A[11]
    u = (A[0] * x + A[3] * y + A[6] * z +A[9]) / w
    v = (A[1] * x + A[4] * y + A[7] * z + A[10]) / w

    result += tex2D ( tRef, (u + 0.5), (v + 0.5)) / $w^2$.

    //repeat for projection 2 with A[12-23] and tRef2

    //repeat for projection 3 with A[24-35] and tRef3

    // mapping voxel (x,y,z) to projection 4 and backproject

    w = A[38] * x + A[41] * y + A[44] * z + A[47]
    u = (A[36] * x + A[39] * y + A[42] * z +A[45]) / w
    v = (A[37] * x + A[40] * y + A[43] * z + A[46]) / w

    result += tex2D ( tRef4, (u + 0.5), (v + 0.5)) / $w^2$
    f_L[k * $L^2$ + row * L + col] = result
END
```

Figure 4: Pseudocode for the fully optimized configuration.

We found that operating on more than four projections led to significant performance degradation. This is because there is a decrease in occupancy. We found that the more computations the kernel was performing, the more registers the compiler was using. At five projections, the occupancy fell below 0.5. This offset any performance gains realized earlier in this section.

VII. Optimizations

Many of the optimization techniques we employed yielded relatively small performance gains. Some optimization techniques yielded no performance gains. The most effective techniques were those that attempted to optimize memory bandwidth. In fact, some of the most effective optimizations were “common sense” optimizations. One such optimization was simply copying the result from the GPU only after the last projection was operated on.

Pre-fetching was another important technique we used. The pseudo code in figures 3 and 4 both employ pre-fetching. The latency of the read at the top of the loop can be hidden by the computation following it. The compiler is not sophisticated enough to determine whether pre-fetching will be an effective optimization, and so it is up to the developer to make it explicit. The naïve configuration of figure 2 does not employ pre-fetching. With this configuration, there is a global memory read followed by a memory write. We found that it is usually best to try and separate memory accesses to give the GPU an opportunity to hide the latency.

Another effective technique we used was to allocate the result array, **f_L**, as page-locked. Page locked memory forces the operating system to store this data on one contiguous page of memory. This has the benefit of eliminating the need for page swaps by the operating system. By allocating memory as page-locked, memory copies from the device to host, and host to device, are faster. Page locked memory reduced the total runtime by 0.8 seconds. It is important to note, however, that excessive use of page-locked memory will reduce overall system performance.

There were optimizations that we tried that gave us no performance benefits. Techniques like loop unrolling and fast-math were not effective. This is because these techniques focus on reducing the total number of instructions. Since this is a memory intensive application, there were not enough instructions to begin with to hide the memory latency. Any technique that increases instruction throughput would see little to no performance impact.

VIII. Results

We reconstructed two volumes, 256^3 and 512^3. We will show the runtimes of the various configurations presented throughout this paper. We will also compare our results to the current best known implementation from [5]. The experiments were conducted on an NVIDIA GTX 480 GPU, programmed with CUDA 3.0 runtime API and with an Intel Core 2 Duo CPU @ 2.66GHz. We built the program in 32bit mode.

Table I shows the runtimes of the various configurations we presented. We can clearly see the effectiveness of our ASIC implementation. When compared to the naïve implementation, we see a speed up of 2.19 for the 256^3 volume and a speed up of 3.9 for the 512^3 volume. The fully optimized configuration yields a speed up of 1.3 for the 256^3 volume and a speed up of 1.78 for the 512^3 volume when compared to the ASIC implementation. Figure 5 presents an axial slice of the reconstructed image. We can clearly see the image even though there is some noise. The error column of table I is the mean squared error measured in Hounsfield units squared. Our results are compared with the "Gold Standard" results from the RabbitCT website. The error shown in table I is due to floating point inaccuracies. More information about how the error is calculated can be found in [8].

TABLE I
RUN TIMES OF THE THREE CONFIGURATIONS

Configuration	Volume	Total (s)	Mean (ms)	Error (HU^2)	Speed-Up	GUPS
Naïve	256^3	7.77	15.66	8.04	N/A	0.99
ASIC	256^3	3.53	7.13	8.07	2.19	2.19
Fully Opt.	256^3	2.71	5.47	8.07	1.3	2.86
Naive	512^3	42.6	86.08	8.04	N/A	1.45
ASIC	512^3	10.8	21.82	8.07	3.9	5.73
Fully Opt.	512^3	6.07	12.25	8.07	1.78	10.2

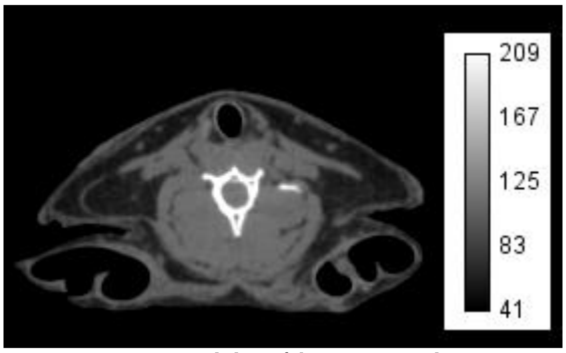

Figure 5: Axial slice of the reconstructed image.

Figure 6 presents the global memory throughput of the three implementations described in this paper. We can see that the fully optimized configuration does not have the highest throughput. We suspect that the occupancy is to blame for this. The occupancy for the ASIC implementation is 1, whereas the occupancy for the fully optimized implementation is 0.66. However, the techniques presented in Section VI make up for the decrease in bandwidth by decreasing the total number of memory accesses.

Table II compares the fully optimized configuration against the best known implementation of [9]. The table shows that our optimized configuration is 1.4 times faster for the 256^3 volume and 2.29 times faster for the 512^3 volume. It is important to note, however, the difference in hardware. The GPU we used is about 1.1 times faster than the GPU used by the top ranked implementation. We estimate that if our implementations were run on the same hardware, our implementation would still be about 2 times faster.

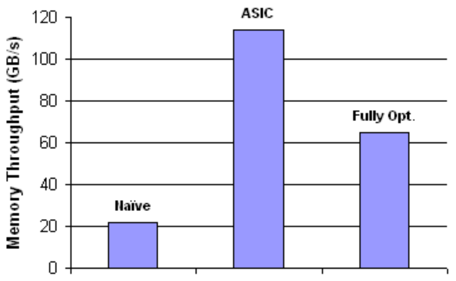

Figure 6: Global memory bandwidth for the naïve, ASIC, and fully optimized implementations.

TABLE II
RUN TIMES OF THE BEST KNOWN AND FULLY OPTIMIZED CONFIGURATIONS

Configuration	Volume	Total (s)	Mean (ms)	Error (HU^2)	Speed-Up
Best Known	256^3	3.843	7.75	8.071	N/A
Fully Opt.	256^3	2.713	5.47	8.071	1.4
Best Known	512^3	13.94	28.11	8.078	N/A
Fully Opt.	512^3	6.076	12.25	8.078	2.29

IX. CONCLUSIONS

We presented a method and analysis of an FDK based back-projection implementation. Many of our techniques can be used in almost any GPU accelerated application. Experimentation to identify the bottlenecks and trying different methods to address them was critical in finding the speed-ups.

REFERENCES

[1] L. Feldkamp L. Davis, J. Kress, "Practical cone beam algorithm," *J. Opt. Soc. Am.* A **1** pp**.** 612–9, 1984.
[2] H. Scherl, B. Keck, M. Kowarschik, J. Hornegger, "Fast GPU-based CT reconstruction using the Common Unified Device Architecture(CUDA)," IEEE Medical Imaging Conference, 6: 4464-4466, Honolulu, HI, 2007.
[3] Z. Zheng, K. Mueller "Cache-Aware GPU Memory Scheduling Scheme for CT Back-Projection," IEEE Medical Imaging Conference, Oct. 2010.
[4] Y. Okitsu, F. Ino and K. Hagihara. "High-Performance Cone Beam Reconstruction Using CUDA Compatible GPUs," *Parallel Computing*, 36(2-3):129-141, 2010.
[5] P. Noël, A. Walczak, J. Xu, J. Corso, K. Hoffmann, S. Schafer, "GPU-based cone beam computed tomography ," *Computer Methods and Programs in Biomedicine,* 98(3):271-277, 2010.
[6] M. Kachelrieß, M. Knaup, and O. Bockenbach, "Hyperfast parallel-beam and cone-beam backprojection using the Cell general purpose hardware," Med. Phys. 34(4):1474-1486, 2007.
[7] X. Jia, Y. Lou, R. Li , W.-Y. Song W, S.-B. Jiang, "GPU-based fast cone beam CT reconstruction from undersampled and noisy projection data via total variation," Med. Phys. 37(4):1757-1760, 2010.
[8] C. Rohkohl, B. Keck, H. G. Hofmann and J. Hornegger, "RabbitCT---an open platform for benchmarking 3D cone-beam reconstruction algorithms," Medical Physics, 36:3940, 2009.
[9] http://www5.informatik.uni-erlangen.de/research/projects/rabbitct/
[10] B. Cabral, N. Cam and J. Foran, "Accelerated volume rendering and tomographic reconstruction using texture mapping hardware," in Proc. of Symp. on Volume Visualization, pp. 91-98, 1994.
[11] K. Mueller "Rapid 3D cone-beam reconstruction with the Simultaneous Algebraic Reconstruction Technique (SART) using 2D texture mapping hardware," IEEE Trans. on Medical Imaging, 19(12):1227-1237, 2000
[12] F. Xu and K. Mueller, "Accelerating popular tomographic reconstruction algorithms on commodity PC graphics hardware" IEEE Trans. on Nuclear Science, 52(3):654-663, 2005
[13] F. Xu and K. Mueller, "Real-time 3D computed tomographic reconstruction using commodity graphics hardware," Physics in Medicine and Biology, 52(12):3405-3420, 2007.
[14] http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUDA_C_Programming_Guide.pdf