Rapid Rabbit: Highly Optimized GPU Accelerated Cone-Beam CT Reconstruction

Eric Papenhausen and Klaus Mueller. *Senior Member, IEEE*

***Abstract* – Graphical processing units (GPUs) have become widely adopted in the medical imaging community. The parallel SIMD nature of GPUs maps perfectly to many reconstruction algorithms. Because of this, it is relatively straightforward to parallelize common reconstruction algorithms (e.g. FDK backprojection). This means that significant performance improvements must come from careful memory optimizations, exploiting ASICs and a few other tricks to boost instruction throughput. We present optimizations that build off of previous work to optimize a GPU accelerated FDK backprojection implementation using the RabbitCT dataset.**

Index Terms—High Performance, GPU, CT reconstruction

I. INTRODUCTION

Within the last ten years, GPUs have become widely accepted in the medical imaging community. This is driven by the relatively low cost of graphics hardware, as well as C like APIs (i.e. CUDA and OpenCL) that were introduced to make GPU programming simpler. GPU manufacturers have embraced the scientific computing use of their graphics cards and are constantly updating the architecture to provide new avenues for optimization.

We make use of the RabbitCT framework [3] which was developed to provide a standardized test of the performance of an FDK backprojection implementation. It comes with a dataset and an executable that runs a backprojection implementation, performs timing, and measures the accuracy of the reconstruction. Since each implementation will reconstruct the same dataset under the same conditions, it becomes very simple to compare different implementations. Recent work has used this framework to help develop highly optimized backprojection implementations. In this paper, we build off the work of [2] to further improve the performance of a GPU accelerated backprojection implementation.

II. RELATED WORK

The first set of major optimizations was presented in our previous work reported in [2]. The parallelization strategy was to launch 512^2 threads and have each thread compute an array of 512 voxels in the Z direction. We gained a significant performance improvement through memory optimizations. We used texture memory to store the projections. By storing the projections this way, we made use of the GPUs ASIC for bilinear interpolation. In order to reduce the total number of global memory accesses, we backprojected multiple projections per GPU kernel call. Figure 1 shows the pseudo-code of this implementation. We have since extended this implementation to include CUDA streams to hide CPU-GPU transfer latency.

```
FOR each voxel along z-axis
  FOR each projection I
   Fetch interpolated value
   Add value to volume
  END
```

Figure 1: Pseudo-code of optimizations presented in [2]

III. OPTIMIZATIONS

In this section we provide a number of additional optimizations that we implemented. We start with an implementation that included the optimizations that were presented in the previous section. This baseline implementation is already quite fast. The optimizations that we describe merely squeeze performance out of an already highly optimized implementation.

A. RSQRT

The baseline implementation was highly optimized in its memory accesses. This made it primarily bound by its instruction throughput. The main source of this bottleneck was the computation of homogeneous coordinates. Figure 2a shows a pseudo-code of this computation. The main source of this bottleneck was the division by *w*. The division operator is extremely slow and in a highly optimized implementation, there are simply not enough instructions to hide the latency this operator. Fortunately, we discovered that we can approximate the division operator by using a technique known as the fast inverse square root [1].

The fast inverse square root works by exploiting the way bits are formatted in a floating point number. By passing the square of some variable, we can get the inverse of that variable with the inverse square root function. Figure 2b shows the homogeneous coordinate calculation with the inverse square root. The CUDA API has an implementation of the inverse square root function that we utilize.

Eric Papenhausen and Klaus Mueller are with the Visual Analytics and Imaging Lab, Computer Science Department, Stony Brook University, Stony Brook, NY 11777 USA (phone: 631-632-1524; e-mail: {epapenhausen, zizhen, mueller}@cs.sunysb.edu). Klaus Mueller is also with SUNY Korea, Songdo, Incheon Korea.

TABLE I: RESULTS OF OPTIMIZATIONS PRESENTED (WALL CLOCK TIME)

Implementation	Thumper	Baseline	+RSQRT	+Transpose	+Multi-Threaded	+Atomics
Timing	0.993 s	2.2 s	1.01s	0.967 s	0.951 s	0.921 s
GUPS	67.7	30.3	65.9	68.8	70.2	72.3

(a) $u = (a_0x + a_3y + a_6z + a_9) / w$
$v = (a_1x + a_4y + a_7z + a_{10}) / w$
$w = a_2x + a_5y + a_8z + a_{11}$

(b) $u = (a_0x + a_3y + a_6z + a_9) * w'$
$v = (a_1x + a_4y + a_7z + a_{10}) * w'$
$w = a_2x + a_5y + a_8z + a_{11}$
$w' = rsqrt(w * w)$

Figure 2: (a) Homogeneous coordinate calculation with division and (b) using the inverse square root function.

B. Transpose

As our baseline implementation would execute, we noticed that there was a performance dip between the third and eighth kernel execution. Each kernel execution would last approximately 50 milliseconds at the beginning. The kernel executions would gradually get slower until it reached around 65 milliseconds, and then get faster toward the end. We realized that the dip in performance was because the cache locality was worse in the middle than it was at the beginning and end of the execution. Figure 3 illustrates this.

In an attempt to increase the cache hit rate, we transpose the volume for the kernel executions with low texture cache locality. This is achieved by simply swapping the *x* and *y* indices of each thread. This led to a 50-80 millisecond reduction in computation time. There is one caveat, however. We need to transpose it back to the proper orientation. We used a similar implementation to [4] for the transpose. This took approximately 20 milliseconds. We still have a net gain, however, of 30-50 milliseconds in performance.

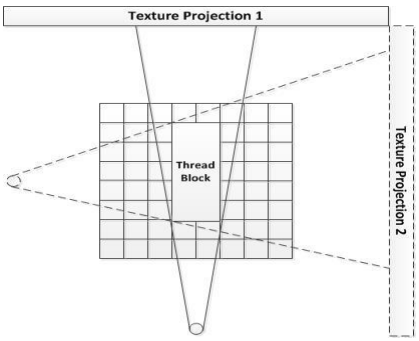

Figure 3: The footprint of the dotted rays is much larger than the footprint of the solid rays, leading to a lower cache locality.

C. Atomics

One unusual optimization that we exploited was using atomics. When programming on the GPU, a general rule of thumb is to avoid using atomics. Atomic operations interupt parallelism and so should only be used in cases where it is unavoidable. This line of thinking has changed with the NVIDIA Kepler architecture.

With the Kepler architecture, atomics are implemented in an ASIC. Furthermore, atomic operations are executed asynchronously with the calling thread. Because of these features, we accumulate the results into the volume using atomic operations. We know that this will be a fast operation since there are no read/write collisions between threads. The asynchronous nature of atomics guarantees that each thread will not have to stall after a write to global memory. This optimization gave us a 3-5 millisecond speed up per kerenel execution.

D. Multi-Threading

In order for us to take advantage of CUDA streams, we have page-lock the projection memory. Page-locked memory, however, is a scarce resource. Therefore, we cannot simply page-lock all the projections at the beginning. In our implementation, we would backproject 64 projections before loading another 64 projections into the page-locked memory. Each memory copy costs approximatly 37 milliseconds. By performing the memory copy in another thread, we could hide some of this latency.

Our multi-threaded approach is to have two buffers. One buffer is the active buffer (i.e. the buffer containing the data that is copied to the device and backprojected). The second buffer is the copy buffer (i.e. the buffer that the second thread copies the next 64 projections to). Once the projections are backprojected, the copy buffer becomes the active buffer, and the current active buffer becomes the next iteration's copy buffer.

IV. RESULTS AND CONCLUSIONS

Our experiments were performed on the NVIDIA GeForce GTX 680 GPU. In Table 1, we compare the results of our optimizations with the current best time (i.e. the algorithm named Thumper) on the RabbitCT rankings. Note that due to the asynchronous nature of CUDA the RabbitCT time is not fully accurate and so we report the wall clock time. These timings represent the reconstruction time for a 512^3 voxel volume, reconstructed from 496 projections. The performance is measured in seconds and giga-updates per second (GUPS). We can see that the optimizations presented lead to nearly a 10% speed-up over the baseline. We observed a root mean squared error of 0.157 HU. This error is similar to the results measured in the RabbitCT rankings.

Although the performance gains were not dramatic, we believe there is not a lot of room for improvement on the current generation of GPUs. As new GPU architectures are introduced, new features will be added that may be exploited to increase performance. The optimizations presented in this paper, however, are likely to remain relevant for future GPU architectures.

ACKNOWLEDGEMENTS

Partial funding was provided by NSF grant IIS 1117132 and the Korean Ministry of Science, ICT and Future Planning and NIPA.

REFERENCES

[1] C. Lomont, "Fast inverse square root," Department of Mathematics, Purdue University, Tech. Rep. Feb. 2003

[2] E. Papenhausen, et.al, "GPU-Accelerated Back-Projection Revisited: Squeezing Performance by Careful Tuning," Workshop on High Performance Image Reconstruction, Potsdam, Germany, July 2011.

[3] C. Rohkohl, et. al, "RabbitCT---an open platform for benchmarking 3D cone-beam reconstruction algorithms," Med. Phys., 36:3940, 2009.

[4] G. Ruetsch, P. Micikevicius, "Optimizing Matrix Transpose in CUDA", NVIDIA Technical Report, 2009