Systematic Performance Optimization of Cone-Beam Back-Projection on the Kepler Architecture

Timo Zinßer and Benjamin Keck

***Abstract*—Filtered back-projection algorithms are widely used for the reconstruction of volumetric data from cone-beam projections in interventional C-arm computed tomography. Furthermore, general-purpose GPUs have become a popular tool for accelerating the reconstruction during time-critical clinical procedures. In this work, we focus on the systematic performance optimization of cone-beam back-projection on the latest architecture of CUDA-enabled GPUs. Our optimization approach is based on the identification of the major performance bottleneck through the analysis of specifically modified kernels.**

Our main contribution is a smart restructuring of the back-projection algorithm that facilitates the simultaneous processing of a large number of projections and improves the hit rate of the texture cache at the same time. We use the well-known RabbitCT benchmark to demonstrate the outstanding performance of our implementation on a single Kepler-based GeForce GTX 680 GPU. Our implementation performs the back-projection of 496 input projections onto a cubic 512^3 volume in less than one second, which is three times as fast as the best competing implementation. Our back-projection implementation is also able to reconstruct a cubic 1024^3 volume in about six seconds, which is six times as fast as the best competing implementation known to us.

***Index Terms*—computed tomography, CUDA, FDK, GPGPU**

I. Introduction

There are cone-beam back-projection implementations for a wide range of hardware platforms, including the Cell broadband engine [1], [2], multi-core CPUs [3], [4], and general-purpose GPUs [5]–[7]. Performance comparisons of several implementations have been provided in [1]–[3]. However, the use of varying data sets and diverse reconstruction parameters precludes a meaningful comparison of the existing implementations. This problem has been tackled with the creation of the RabbitCT platform [8], which provides a standardized framework for comparing both the accuracy and the performance of cone-beam back-projection algorithms.

According to [9], filtered back-projection was the first non-graphics compute application to be successfully accelerated on a dedicated GPU. In the meantime, GPUs have evolved into programmable many-core processors, and development platforms like the CUDA framework [10] have simplified the implementation of GPU-accelerated algorithms considerably. Okitsu *et al.* present a comprehensive overview of techniques for the efficient implementation of cone-beam back-projection on CUDA-enabled GPUs in [6]. Their most important contribution is the substantial reduction of memory accesses by processing several projections at a time. Papenhausen *et al.* describe a back-projection implementation that is optimized for Fermi-based GPUs with CUDA support in [7].

T. Zinßer and B. Keck are with Siemens AG, Healthcare Sector, Imaging & IT Division, P.O. Box 1266, D-91294 Forchheim, Germany. Corresponding author: Timo Zinßer, E-mail: timo.zinsser@siemens.com.

As we use the RabbitCT benchmark to evaluate our work, we shortly describe the corresponding data set in Section II. We then discuss several important aspects of GPU computing in Section III. Our approach for systematic performance optimization, as well as the resulting optimized implementation, are presented in Sections IV and V. We analyze the performance of our cone-beam back-projection implementation in Section VI. Finally, Section VII concludes this work.

II. Problem Description

The input data set of the RabbitCT benchmark consists of $N = 496$ projections with a width of $S_u = 1248$ pixels and a height of $S_v = 960$ pixels. For every projection n, the data set also contains a projection matrix $\boldsymbol{P}_n \in \mathbb{R}^{3\times4}$, which describes the transformation from the world coordinate system to the detector coordinate system. An illustration of the acquisition geometry can be found in [4]. Basically, the detector rotates on a circular short-scan trajectory around the z-axis of the world coordinate system, which is also the z-axis of the reconstructed volume. This axis is roughly perpendicular to the u-axis of all projections, as well as roughly parallel to their v-axis.

The task of a back-projection algorithm in the RabbitCT benchmark is the reconstruction of a cubic volume with a side length of 256 mm. The benchmark defines three problem sizes, which correspond to volumes with a side length of 256, 512, or 1024 voxels, respectively. During the kernel optimization in Section IV, we focus on the 512^3 volume. As every projection has to be back-projected onto every voxel, the reconstruction of the 512^3 volume requires approximately 66.6×10^9 voxel updates. This value is important for computing a common alternative performance measure, the giga-(voxel)-updates per second (GUPS). Please take note that we strictly differentiate between decimal prefixes (1 gigabyte = 1 GB = 10^9 bytes) and binary prefixes (1 gibibyte = 1 GiB = 2^{30} bytes) to prevent unnecessary ambiguities in this work.

III. GPU Computing

In our experience, the full performance potential of any new hardware platform can only be realized by specifically optimizing the implementation of the deployed algorithms. The main hardware platform of this work is the Kepler-based GeForce GTX 680 GPU, which is compared to its predecessors

TABLE I
RECENT GENERATIONS OF HIGH-END GPUS WITH CUDA SUPPORT

GPU	GTX 280	GTX 480	GTX 580	GTX 680
Architecture	Tesla	Fermi	Fermi	Kepler
Performance [GFLOPS]	622.1	1345.0	1581.1	3090.4
Texture fillrate [GT/s]	48.2	42.0	49.4	128.8
Bandwidth [GB/s]	141.7	177.4	192.4	192.3

in Table I. Evidently, the arithmetic throughput of these GPUs has increased with every generation. It is important to note that the specified peak performance is only achieved for fused multiply-add operations. For other operations, the performance is reduced at least by a factor of two. The texture fillrate has remained almost constant for several generations, but it has more than doubled for the GeForce GTX 680. In contrast to this, the memory bandwidth has all but stagnated in the latest generation. As a consequence, the ratio between the texture fillrate and the memory bandwidth has also more than doubled. In order to save memory bandwidth and fully utilize its texture units, the GeForce GTX 680 buffers texture data in the read-only caches of its streaming multiprocessors as well as in its unified L2 cache. For convenience, we refer to the combination of these caches as texture cache.

We use the CUDA framework for implementing GPU-based algorithms. An overview of basic concepts of GPU computing can be found in [9]. For advanced topics, we recommend the documentation of the CUDA framework itself [10]. One important aspect of optimization is the identification of the current performance bottleneck. Typically, the performance of an algorithm is either limited by the instruction throughput, the memory bandwidth, or the texture fillrate. In contrast to this, latencies are usually not a problem, as long as the algorithm exposes enough thread-level or instruction-level parallelism.

Compared to a sequential CPU algorithm, the execution order of a highly parallel GPU algorithm is more complicated and less deterministic. The following overview identifies various levels of temporal cohesion on a Kepler-based GPU:

- The 32 threads of one warp run in lockstep. They can communicate via very efficient shuffle instructions.
- One or more warps form a thread block. The threads in one thread block can communicate via shared memory. These threads can also be synchronized explicitly.
- One or more threads block are executed on a streaming multiprocessor. These thread blocks share the L1 cache and the read-only cache of the multiprocessor. The ratio between the actual number of threads on a multiprocessor and the theoretical maximum is called occupancy.
- In general, only a subset of all thread blocks fits onto the available multiprocessors at the same time. This subset is called a wave. The thread blocks of a wave share the unified L2 cache of the GPU.

Iterative loops inside a kernel exhibit less temporal cohesion than the threads of a warp, and may have less temporal cohesion than the threads of a thread block, if these threads are synchronized after every iteration. The execution order of instructions in different thread blocks is not defined by the CUDA programming model.

```
compute position of first voxel
for $I$ input projections do
    compute homogeneous detector coordinates $\boldsymbol{q}[i]$ of first voxel
end
for $K$ consecutive voxels along the $z$-axis do
    zero-initialize sum $s$ of weighted back-projected values
    for $I$ input projections do
        dehomogenize detector coordinates $\boldsymbol{q}[i]$
        compute back-projected value by texture fetching
        update sum $s$ of weighted back-projected values
        update homogeneous detector coordinates $\boldsymbol{q}[i]$
    end
    update volume at current voxel with computed sum $s$
    (optionally) synchronize threads in thread block
end
```

Fig. 1. Pseudocode for cone-beam back-projection kernel A

IV. KERNEL OPTIMIZATION

The cone-beam back-projection kernel presented in Fig. 1 constitutes the starting point of our performance optimization. The structure of this kernel is based on the structure of what is referred to as the fully optimized configuration in [7]. In our kernel, one thread updates K voxels with the weighted back-projected values of I projections. Every thread block consists of $B_x \times B_y$ threads. The voxels updated by a single thread block constitute a rectangular tile in the respective consecutive xy-slices of the volume. A volume with $S_x \times S_y \times S_z$ voxels is processed by a grid of $(S_x/B_x) \times (S_y/B_y) \times (S_z/K)$ thread blocks. We use layered textures to simplify the texture fetching for several projections. We also store the projection matrices in constant memory, which is backed by the read-only cache. Finally, we specify the number of projections as a template parameter, which allows the compiler to automatically unroll the corresponding loops of kernel A.

In this section, we ignore all data transfers involving the host and focus on the computation times of the kernels for the 512^3 volume. In order to identify the performance bottleneck of a configuration, we measure three additional computation times. In the first step, we reduce the voxel size from 0.5 mm to 10^{-6} mm. As a result, all computed detector coordinates are virtually identical, and the hit rate of the texture cache rises to almost one hundred percent. In the second step, we disable the texture fetching completely. In the third step, we also turn off the volume update, which removes the memory accesses and leaves only the arithmetic and control flow instructions. It is vital that these modifications do not allow the compiler to eliminate more code than intended. As these modifications also tend to reduce the register count, we allocate a suitable amount of shared memory to retain the occupancy of the original kernel. Using the letters I(nstruction), M(emory), and T(exture), we label the corresponding additional computation times as $\mathrm{I}\,|\mathrm{M}|\,\mathrm{T}$, $\mathrm{I}\,|\mathrm{M}|\,-$, and $\mathrm{I}\,|-|\,-$ in Table II.

In our first test, kernel A processes one projection at a time. Each thread updates one voxel in every xy-slice of the volume. The specified tile width $B_x = 32$ ensures that the volume updates are performed by fully coalesced memory transactions. Nevertheless, the first row of Table II clearly shows that the memory transfer takes much longer than the computation of the arithmetic instructions. In addition to that, the computation

TABLE II
PERFORMANCE BOTTLENECK ANALYSIS FOR DIFFERENT KERNEL CONFIGURATIONS

Test	Kernel	Sync	I	K	B_x	B_y	Occupancy	I |–|–	I |M|–	I |M|T	Time	GUPS
1	A	no	1	512	32	8	1.000	1091 ms	4710 ms	4707 ms	9141 ms	7.3
2	A	no	4	512	32	8	0.750	575 ms	1199 ms	1196 ms	7802 ms	8.5
3	A	yes	4	512	32	8	1.000	554 ms	1185 ms	1169 ms	2710 ms	24.6
4	A	no	4	8	32	8	0.750	685 ms	980 ms	1085 ms	1990 ms	33.4
5	B	—	4	8	32	8	0.875	542 ms	989 ms	1179 ms	1527 ms	43.6
6	B	—	8	4	16	16	0.750	528 ms	725 ms	966 ms	1296 ms	51.4
7	B	—	16	4	16	32	0.750	506 ms	550 ms	826 ms	1051 ms	63.4
8	B	—	32	4	16	32	0.750	489 ms	494 ms	756 ms	969 ms	68.7

```
compute position of first voxel
for $K$ consecutive voxels along the $z$-axis do
    zero-initialize sum $s[k]$ of weighted back-projected values
end
for $I$ input projections do
    compute homogeneous detector coordinates $\boldsymbol{q}$ of first voxel
    for $K$ consecutive voxels along the $z$-axis do
        dehomogenize detector coordinates $\boldsymbol{q}$
        compute back-projected value by texture fetching
        update sum $s[k]$ of weighted back-projected values
        update homogeneous detector coordinates $\boldsymbol{q}$
    end
end
for $K$ consecutive voxels along the $z$-axis do
    update volume at current voxel with computed sum $s[k]$
end
```

Fig. 2. Pseudocode for cone-beam back-projection kernel B

time of the kernel is almost doubled by the cache misses of the texture fetching. When we process four projections in one kernel, the memory transfer size is reduced considerably. The compute-only kernel also runs much faster, because the number of integer-based index computations is minimized as well. However, the time penalty induced by the cache misses of the texture fetching remains very high.

In the third test, we activate the optional synchronization as indicated in Fig. 1. This change prevents the divergence of the threads in one thread block with respect to the loop over the voxels along the z-axis. As a result, the texture fetching is accelerated considerably and the computation time is reduced by about 65 percent. The configuration of test 3 results in a total of 16 waves of thread blocks, which iterate through the volume along the z-axis one after another. In test 4, we relocate the large scale movement along the z-axis from the loop inside the kernel to the third dimension of the grid of thread blocks. On the whole, the 1024 generated waves move through the volume along the z-axis only once, which improves the hit rate of the texture cache even more.

In all tests with kernel A, the cache misses of the texture fetching constitute the major performance bottleneck. As the innermost loop of this kernel iterates over different projections for $I > 1$, the corresponding textures continuously contend for the limited amount of cache memory. Furthermore, the memory transfers for the volume update take longer than the computations. This problem could be solved by increasing the number of projections I, but this approach only exacerbates the first problem. We propose to solve both problems by reversing the order of the two nested loops in kernel A. The result of this restructuring is illustrated in Fig. 2. In kernel B, we specify both I and K as template parameters. All iterations of the innermost loop of this kernel access the same texture. While kernel A uses $3I$ registers to store the homogeneous detector coordinates, kernel B requires K registers for buffering the computed volume updates. Consequently, our proposed kernel is able to process a very large number of projections.

In test 5, we replace kernel A with kernel B, but keep all other parameters identical. We clearly observe an improved hit rate of the texture cache. In the following three tests, we increase the number of projections I and tune the other parameters to obtain minimal computation times. For $I = 32$ projections, the instruction throughput is not the bottleneck and the impact of the memory transfer is negligible. The computation time of the I |M| T modification indicates that our proposed kernel reaches more than 68% of the theoretical texture fillrate. The impact of the cache misses of the texture fetching has also been reduced, resulting in a total computation time of less than one second for the 512^3 volume.

V. DATA TRANSFER OPTIMIZATION

For a useful comparison of our GPU-based implementation with other hardware platforms, the data transfers between the host and the GPU have to be taken into consideration. The practically relevant data transfers consist of the upload of the input projections and the download of the reconstructed volume. The reconstruction of the 512^3 volume of the RabbitCT benchmark results in the transfer of 2779 MiB of data, which takes about half a second on our system. In order to hide the additional time required for the described data transfers, we use the ability of our GPU to overlap kernel execution and data transfer. To this end, the CUDA API allows asynchronous kernel launches and provides asynchronous memcopy functions. However, there are no asynchronous API functions for binding textures, which complicates both the memory management and the texture handling in our implementation.

The timeline in Fig. 3 illustrates the data transfers and kernel executions during the reconstruction of the 512^3 volume. As the first data transfer cannot be overlapped with any kernel launch, we start with $I = 8$ projections to keep the size of this data transfer small. We add eight more projections in every subsequent data transfer until we reach the optimal value of $I = 32$ projections. As a second optimization, we divide the reconstructed volume into two parts, which consist

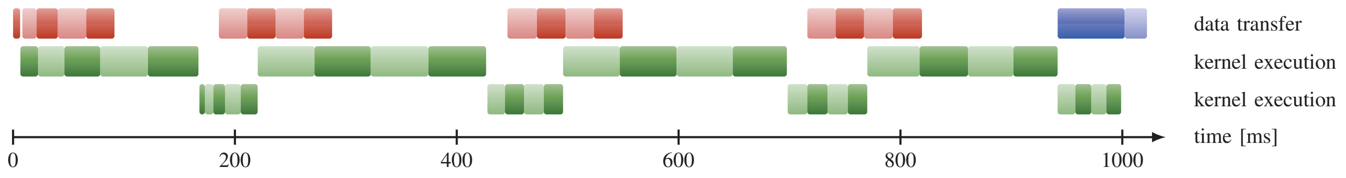

Fig. 3. This figure displays the timeline of the reconstruction of the 512^3 volume. The first line represents the data transfers, which comprise the upload of the projections and the download of the volume. The other two lines illustrate the reconstruction of the first and the second part of the volume, respectively. Please note that the total computation time in this timeline is larger than one second due to profiling overhead.

TABLE III
COMPARISON OF SELECTED CONE-BEAM BACK-PROJECTION IMPLEMENTATIONS LISTED ON THE RABBITCT HOMEPAGE

Volume	Implementation	Type	RMSE	Time	GUPS
512^3	fastrabbitEX [4]	CPU	—	7.45 s	8.94
	RapidRabbit [7]	GPU	—	2.98 s	22.3
	Thumper [this work]	GPU	0.021 HU	0.99 s	67.7
1024^3	fastrabbitEX [4]	CPU	—	43.8 s	12.2
	CERA [-]	GPU	—	36.1 s	14.7
	Thumper [this work]	GPU	0.021 HU	6.04 s	88.2

of 384 and 128 xy-slices, respectively. In combination with the buffering of a certain number of projections on the GPU, this optimization makes it possible to overlap the download of the first part of the volume with the reconstruction of the second part of the volume.

Our implementation also works with the 1024^3 volume of the RabbitCT benchmark. However, this volume has a size of 4096 MiB, which is twice as large as the device memory of our GPU. As there are no data dependencies between different voxels or between different projections, we are free to adapt the high-level data flow of our implementation accordingly. We have chosen to upload the first half of the projections onto the GPU and stream the 1024^3 volume using two buffers with a size of 256 MiB each. We repeat this process for the second half of the projections. Performance measurements for this volume size are provided in the next section.

VI. EXPERIMENTAL RESULTS

Our presented cone-beam back-projection implementation was evaluated on a GeForce GTX 680 GPU with 2048 MiB RAM using version 4.2 of the CUDA framework. In Table III, we compare the obtained results to the best competing implementations listed on the RabbitCT homepage [11]. Our implementation, alias Thumper, has a total computation time of less than one second for the 512^3 volume. This is three times as fast as the best competing implementation RapidRabbit. As both implementations were tested on the same GPU model, the performance difference can be fully attributed to our proposed optimizations. Furthermore, our implementation has a total computation time of about six seconds for the 1024^3 volume. This is six times as fast as the best competing implementation, which uses a slower Fermi-based Tesla C2070 GPU.

The fastest CPU-based implementation in the RabbitCT benchmark is called fastrabbitEX. Although a workstation with 40 CPU cores was used to evaluate fastrabbitEX, our implementation is more than seven times as fast for both volume sizes. Due to an error in the reference volume of the benchmark, we are unable to specify the accuracy of the competing implementations in Table III. When we use the corrected reference volume in floating-point format to assess the accuracy of our implementation, we observe a root mean square error of only 0.021 HU for both volume sizes.

VII. CONCLUSION

Using a systematic performance optimization approach, we identify the bottlenecks of a state-of-the-art cone-beam back-projection kernel. Our main contribution is a restructuring of the kernel that deals with the two most prominent performance bottlenecks all at once. Our implementation is three times as fast as the best competing implementation for the clinically relevant 512^3 volume of the RabbitCT benchmark. Although we focus on the optimization of a cone-beam back-projection kernel for GPUs of the Kepler architecture, our optimization approach is applicable to a wide range of GPU-based algorithms. Furthermore, cursory tests with other CUDA-enabled GPUs have confirmed the portability of our optimizations.

REFERENCES

[1] M. Kachelrieß, M. Knaup, and O. Bockenbach, "Hyperfast parallel-beam and cone-beam backprojection using the Cell general purpose hardware," *Medical Physics*, vol. 34, no. 4, pp. 1474–1486, April 2007.
[2] H. Scherl, M. Kowarschik, H. Hofmann, B. Keck, and J. Hornegger, "Evaluation of State-of-the-Art Hardware Architectures for Fast Cone-Beam CT Reconstruction," *Parallel Computing*, vol. 38, no. 3, pp. 111–124, March 2012.
[3] H. Hofmann, B. Keck, C. Rohkohl, and J. Hornegger, "Comparing performance of many-core CPUs and GPUs for static and motion compensated reconstruction of C-arm CT data," *Medical Physics*, vol. 38, no. 1, pp. 468–473, January 2011.
[4] J. Treibig, G. Hager, H. Hofmann, J. Hornegger, and G. Wellein, "Pushing the limits for medical image reconstruction on recent standard multicore processors," *International Journal of High Performance Computing Applications*, 2012, OnlineFirst.
[5] D. Riabkov, X. Xue, D. Tubbs, and A. Cheryauka, "Accelerated cone-beam backprojection using GPU-CPU hardware," in *Proceedings of the Workshop on High Performance Image Reconstruction*, Lindau, Germany, July 2007, pp. 68–71.
[6] Y. Okitsu, F. Ino, and K. Hagihara, "High-Performance Cone Beam Reconstruction Using CUDA Compatible GPUs," *Parallel Computing*, vol. 36, no. 2-3, pp. 129–141, February 2010.
[7] E. Papenhausen, Z. Zheng, and K. Mueller, "GPU-Accelerated Back-Projection Revisited: Squeezing Performance by Careful Tuning," in *Proceedings of the Workshop on High Performance Image Reconstruction*, Potsdam, Germany, July 2011.
[8] C. Rohkohl, B. Keck, H. Hofmann, and J. Hornegger, "RabbitCT - an open platform for benchmarking 3D cone-beam reconstruction algorithms," *Medical Physics*, vol. 36, no. 9, pp. 3940–3944, 2009.
[9] G. Pratx and L. Xing, "GPU computing in medical physics: A review," *Medical Physics*, vol. 38, no. 5, pp. 2685–2697, May 2011.
[10] "NVIDIA CUDA Toolkit Documentation," 2013. [Online]. Available: http://docs.nvidia.com/cuda/index.html
[11] "Homepage of the RabbitCT project," 2013. [Online]. Available: http://www5.cs.fau.de/research/projects/rabbitct