Putting 'p' in RabbitCT - Fast CT Reconstruction Using a Standardized Benchmark

Hannes G. Hofmann, Benjamin Keck, Christopher Rohkohl, and Joachim Hornegger

Pattern Recognition Lab, Friedrich-Alexander University Erlangen-Nuremberg
Martensstr. 3, 91058 Erlangen, Germany
{hannes.hofmann,benjamin.keck,christopher.rohkohl,
joachim.hornegger}@informatik.uni-erlangen.de
http://www5.informatik.uni-erlangen.de/

Abstract. Computational architectures and processors are an ever-changing field of research and development. Standardized and representable problem-dependent tests are required to find the optimal design of a running system. For the demanding problem of 3-D cone-beam tomographic reconstruction no such benchmarks are available to the public. We provide a standardized benchmark environment www.RabbitCT.com which enables the comparison of different hardware and system configurations. In this work we report the first results from a handful of different parallelization approaches. Among them software-based multi-threading, SIMD optimizations and state-of-the-art graphics cards using the CUDA programming environment.

Key words: Back-projection, Benchmark, CBCT, CT, GPU, Multi-core

1 Introduction

Clinical applications require fast 3-D reconstruction of tomographic data. Therefore, means to accelerate the reconstruction is a strong research topic. Currently the most wide-spread algorithms in clinical X-ray CT reconstruction belong to the class of filtered backprojection (FBP), e.g. the FDK method [1] for cone-beam data. Another class of reconstruction algorithms is based on iterative techniques which require multiple iterations of back- and forward projections. The latter can incorporate various corrections and provide superior image quality in certain cases like sparse or irregular data [2]. The fact that their complexity is a multiple of the FDK's might be a reason why this class of reconstruction algorithms is less commonly used in clinical systems.

The backprojection step is complex and bandwidth demanding – yet highly parallelizable. Hence, both classes of reconstruction algorithms can be accelerated well.

A manifold of specialized hardware and software techniques exist which can be applied to a multitude of reconstruction algorithms and acquisition scenarios. However, there exists no public means for comparing the speed and accuracy across different publications. We have realized the need for a standardized benchmark for reconstruction performance. Therefore, we have initiated RabbitCT [3, 4] which fills this gap (see Section 2.4). To test-drive this platform we have implemented a handful of modules with different parallelization approaches. Besides multi-threading using different libraries we present a hand-optimized SIMD implementation, a version combining both techniques and furthermore a GPU implementation using CUDA 2.1.

2 Methods and Materials

Over the last few years it became clear that the performance gain of new processors no longer comes from raising the clock rate. Instead, processor vendors have shifted paradigms and turn the increasing number of transistors into an increasing number of cores. Software developers have to write parallel code now to leverage all the available compute power. Multi-threading libraries (Sect. 2.1) help to utilize multiple cores. To also exploit the SIMD units on recent processors, the code has to be vectorized (Sect. 2.2). And finally, modern graphics cards are real work horses that can be employed by use of CUDA (Sect. 2.3) and the forthcoming OpenGL. This section is concluded by a description of our RabbitCT benchmark and the hardware used for our experiments.

2.1 Multi-Threading Libraries

Several APIs exist which support developers in multi-threaded programming. In this paper we have evaluated two of them, Intel's Threading Building Blocks (TBB) [5] and OpenMP [6]. We chose TBB because it will be supported on Larrabee [7], our next target platform, and OpenMP for its popularity and ease of use.

OpenMP is well established and supported by major hardware and software vendors. Code sections that should be executed in parallel are marked with a preprocessor directive. Thus compiler support is required which is provided with most modern compilers.

The open source project TBB was founded by its main contributor Intel. TBB is implemented as a C++ template library. It does not require compiler support and offers plenty of components for parallel programming.

2.2 Vectorization

Modern CPUs feature so-called vector processing units. Instead of performing one operation on a single element of data they can perform the same instruction on many elements concurrently, called SIMD (Single Instruction, Multiple Data). Current CPUs from AMD and Intel support SSE which can operate on four single precision floating point numbers at once. Future chips, e.g. Larrabee, will extend the SIMD width and be able to process up to 16 single precision elements simultaneously [7]. To utilize SIMD instructions, the programmer has to use intrinsics, which are translated into the corresponding assembler instructions by the compiler. Most modern compilers also try to automatically vectorize code – with varying success. The performance benefit of vectorization is greatest in parts of the program where many data elements are processed in the same way.

2.3 CUDA 2.1

In 2007, NVIDIA introduced their Common Unified Device Architecture (CUDA) [8], which was a fundamentally new programming approach at this time, making use of the unified shader design of the most current Graphics Processing Units (GPUs) from NVIDIA. The programming interface allows to implement an algorithm using standard C language and a few extensions. No knowledge about graphics programming using OpenGL, DirectX, and shading languages is necessary any more.

Since then CUDA evolved, up to CUDA 2.1 in the latest release, offering a stable system. It provides access to many hardware features, for the usage of high performance parallel computation. For example, NVIDIA's QuadroFX 5600, which we used for our experiments, uses 128 stream processors in parallel and can additionally benefit from hardware accelerated features like texture interpolation. Furthermore, CUDA can be used on the latest and future graphics cards by NVIDIA.

Applying the easy to use C-like application programming interface of CUDA, the following two different parts in each CUDA implementation have to be distinguished: A host part, which executes in a CPU thread, and a device part (kernel). The latter is invoked by the controlling CPU thread, but runs in parallel on the GPU device. Due to the fact that the kernel can only operate on the graphics card's memory, the API provides functions for memory management from within the CPU thread.

2.4 RabbitCT

To evaluate different approaches to exploiting parallelism, we implemented several modules for the benchmark suite RabbitCT using different optimization techniques. RabbitCT provides a standardized C-arm CT dataset, problem statements and framework to benchmark the performance of the backprojection step of the FDK algorithm. The dataset consists of $N = 496$ pre-processed projection images $\boldsymbol{I}_n \in \mathbb{R}^{S_x \times S_y}, n = 1 \dots N$ from a C-arm system (Siemens AG, Artis Zee) acquired on a 200° circular short-scan trajectory. The size of a projection image is $S_x = 1248$ pixels in width and $S_y = 960$ pixels in height at an isotropic resolution of $0.32\ \frac{\text{mm}}{\text{pixel}}$.

For each projection image a pre-calibrated projection matrix $\boldsymbol{A}_n \in \mathbb{R}^{3\times4}$ is available. It encodes the perspective projection, in homogeneous coordinates, of a 3-D object point onto the 2-D projection image [9, 10].

The task for RabbitCT modules is the reconstruction of an isocentric cubic volume of 256^3 mm^3. We defined four different problem sizes with different computational costs. The side lengths of the cubic reconstruction are $L \in \{128, 256, 512, 1024\}$ voxels respectively at an isotropic voxel size of $R_L = \frac{256}{L}$ mm.

2.5 Test Hardware

CPU	Sockets × Cores	Clock Speed	L2 Cache	L3 Cache	RAM
Core2Duo T9300	2	2.50GHz	6MB	–	4GB
Core2Extreme QX9650	4	3.00GHz	2 × 6MB	–	4GB
Xeon X7460	4 × 6	2.67GHz	12 × 3MB	4 × 16MB	32GB
Nehalem-EP	2 × 4	2.67GHz	8 × 256KB	2 × 8MB	12GB

Table 1. Specifications of our test systems. Missing in this list is the graphics card, see text for details.

The most important specifications of our test systems are listed in Table 1. Note that the machines with Xeon X7460 (Dunnington) and Nehalem processors were pre-production systems. We expect production hardware to deliver similar performance levels. Missing in this list is the graphics card used for the CUDA benchmarks. We used a NVIDIA QuadroFX 5600 installed in a Fujitsu Siemens Celsius R650 with an Intel Xeon E5410 CPU and 16 GB of main memory.

Nehalem's two major novelties are the QuickPath Interconnect (QPI) which replaces the legacy Front Side Bus (FSB) and the integrated memory controller with support for DDR3 SDRAM. These two result in a substantial increase in main memory bandwidth.

For tests on the Core2Duo (C2D) and Core2Extreme (C2E) systems the MSVC9 compiler was used. The Dunnington and Nehalem systems were benchmarked using both gcc in version 4.3.3 and the Intel compiler in version 11.0.074.

3 Implementation

Algorithm 1 describes the core part of the FDK algorithm. It is executed for every projection image $\boldsymbol{I}_n$. The three inner loops (i, j, k) traverse the volume in x, y and z direction and update each voxel exactly once.

Algorithm 1: The backprojection kernel for the n-th projection image.

```
input  : $\boldsymbol{I}_n, \boldsymbol{A}_n, L, O_L, R_L, f_L$
output: updated reconstruction $f_L$
for $k = 0$ to $L - 1$ do
    $z = O_L + kR_L$;
    for $j = 0$ to $L - 1$ do
        $y = O_L + jR_L$;
        for $i = 0$ to $L - 1$ do
            $x = O_L + iR_L$;
            // Update the volume
            $f_L(i, j, k) += \frac{1}{w_n(x,y,z)^2} \cdot \hat{p}_n(u_n(x, y, z), v_n(x, y, z))$;
        end
    end
end
```

3.1 Multi-Threading

Within one iteration of the projections-loop the FDK algorithm has no competing write accesses. Therefore, any of the x, y, z loops can be split into disjoint ranges and executed in parallel.

To reduce the overhead of thread management we decided to parallelize only the outermost (z) loop. Of course, the actual number of chunks into which the loop is divided is usually equal to N_P, the number of compute cores. This method will scale up to N_z cores, where N_z is the number of voxels in z-direction (128, 256, 512 or 1024). As N_P is typically small compared to N_z, we can be sure that each task is big enough to compensate the start-up overhead. This strategy was used for both the OpenMP and TBB implementation.

In our OpenMP implementation we defined a parallel work-sharing region (`#pragma parallel for`) around the for-loop which iterates in z-direction.

The TBB implementation uses `parallel_for` as well. But due to the design of TBB the kernel had to be wrapped in a C++ class which stores the state of the current task. The TBB scheduler creates one instance of this class per thread and initializes it with the correct boundaries. Then its `operator()` method is called by the scheduler.

3.2 Vectorization

Vectorization is most efficient in code parts where a lot of computation is done. The y and z loops do only increment the coordinates but the actual backprojection, including a bilinear interpolation, happens within the x-loop. It iterates over all voxels in one column (fixed y and z coordinate) and performs the same computation on each of them. This makes it just natural to use SIMD here.

While multi-threading was supported by the libraries, vectorization was done manually. For many parts of the code it was straightforward. However, vectorization is particularly complicated for code sections that contain conditional branches. There is a check if the current voxel is in the Field-of-View (FOV) of the current projection before the projection value is read. If it lies outside, the voxel update is skipped and processing continues with the next voxel. Since SIMD units can not branch independently for individual vector elements all voxels have to be updated. To avoid invalid memory accesses the projected coordinates of a voxel are set to $(0, 0)$ if they lie outside of the projection. After fetching the projection values, the outliers are set to 0. Both measures can be implemented efficiently using multiplicative masks.

Another problem is caused by the geometry. Rays through adjacent voxels do not necessarily end up in neighboring pixels in the projection image. The result are non-linear memory accesses when loading the projection values for a vector of voxels. Therefore, all pixel values have to be fetched in a scalar manner and inserted into vectors which are then used to compute the bilinear interpolation.

3.3 GPU / CUDA 2.1

In our CUDA 2.1 implementation we invoke our backprojection kernel on the graphics device. In CUDA the parallelization is originated due to the partition of the problem into a grid of blocks, while each block consists of a limited number of threads. Each thread of the kernel computes the backprojection of a certain column of the volume in y-direction. This means that the volume is parallel processed for each x-z-coordinate.

This allows to save six multiply-add operations by incrementing the homogeneous coordinates with the appropriate column of the projection matrix for neighboring voxels in y-direction. This approach also reduces the register usage down to 9 registers for a backprojection kernel. The detailed description of the CUDA implementation can be found in [11].

4 Results

We ran benchmarks for volumes of $128^3, 256^3$ and 512^3 voxels. Most emphasis is put on the results of the largest problem as they are relevant for clinical use. Moreover, the start-up overhead has more impact on the overall runtime on smaller volume sizes. Therefore, if not said explicitly we refer to the 512^3 volume.

RabbitCT measures the plain backprojection runtime without reading from and storing to hard disk. We anticipate a situation where we either have sufficient hard disk speed (e.g. using a RAID system) or get the projections on the fly over a fast network link. To accomplish this, we tried to load all projections to main memory before backprojecting all of them. However, due to memory limitations on some systems, we had to split the projections into four sets and process the sets sequentially. For the CUDA implementation, the final transfer of the reconstructed volume from the GPU to the host's main memory is not included in the measurements because typically the volume would be rendered after reconstruction and could therefore stay on the GPU. The times given in this section are the mean backprojection time for one projection. Runtimes are actual measurements, speed-ups comparing different systems are normalized by the CPUs' clock speed.

	Core2Extreme		Nehalem	
Volume Size	Runtime	Factor	Runtime	Factor
128^3	7.20	1.00×	3.25	1.00×
256^3	49.76	6.91×	22.16	6.81×
512^3	400.38	55.59×	170.74	52.46×

Table 2. Mean backprojection time in ms on the Core2Extreme and Nehalem systems for different volume sizes. Speed-ups are relative to the 128^3 volume. Implementation: TBB+SIMD.

Table 2 shows how the runtime changes with different volume sizes. Theoretically the complexity is increased by a factor of 8 when the length of the reconstructed volume is doubled. However, when increasing the volume size from 128^3 to 256^3 voxels, the factor is significantly less than 8. This indicates that the start-up overhead is a substantial part of the total computation time in the 128^3 case. In contrast, the factor is almost 8 when doubling cube length again to 512^3. The problem size of 256^3 is sufficient to hide the overhead.

Implementation	Core2Duo	Core2Extreme	Nehalem	Dunnington
single-threaded	6322.27	5389.05	5041.82	5064.11
SIMD	1789.47	1505.71	1711.64	1694.94
OpenMP	3126.36	1256.71	456.41	229.52
TBB	2683.81	1119.96	447.50	222.32
TBB+SIMD	990.99	400.38	170.74	116.47

Table 3. Mean backprojection time in ms on selected systems for different optimizations. The volume size was 512^3.

Volume Size	CUDA	Dunnington
128^3	4.00	2.70
256^3	10.14	13.55
512^3	34.50	116.47
1024^3	—	931.83

Table 4. Mean backprojection time in ms on the GPU and on the fastest CPUs. A volume of size 1024^3 was not too large to store on the used graphics card.

Table 3 provides an overview over the measurements of all CPU-based implementations with a 512^3 volume on different systems. On the Nehalem and Dunnington systems the Intel compiler was used, on the Core2Duo and Core2Extreme MSVC. The results of the CUDA implementation can be found in Table 4.

Table 5 shows the achieved speed-ups for different optimizations. The speed-up with OpenMP was 18.8× on the Dunnington system and 10.2× on the 8-core Nehalem using gcc. The Intel compiler performed even better achieving 22.1×

Implementation	Core2Duo Speed-up	Core2Extreme Speed-up	Nehalem Speed-up	Dunnington Speed-up
single-threaded	1.00×	1.00×	1.00×	1.00×
SIMD	3.53×	3.58×	2.95×	2.99×
OpenMP	2.02×	4.29×	11.05×	22.06×
TBB	2.36×	4.81×	11.27×	22.78×
TBB+SIMD	6.38×	13.46×	29.53×	43.48×

Table 5. Shows impact of different optimizations on the same CPU, for two different CPUs. Speed-up is relative to single-threaded implementation on same CPU.

Implementation	Compiler	Dunnington Runtime [ms]	Nehalem Runtime [ms]
OpenMP	gcc	269.23	494.95
OpenMP	icc	229.52	456.41
TBB	gcc	224.23	447.12
TBB	icc	222.32	447.50

Table 6. Comparing gcc and the Intel compiler on two systems. The volume size was 512^3. Note the improvement of the OpenMP implementation when using icc.

and 11.1× respectively. As mentioned in Section 3.1 TBB required more effort to implement. But the reward was better performance throughout all our experiments. The speed-up with TBB was 22.6× and 11.3× respectively, also using gcc, and 22.8×, 11.3× using the Intel compiler. When using MSVC or gcc, the advantage of TBB was between 10 and 20%. Even with the Intel compiler's optimized OpenMP implementation, TBB is still $2-3\%$ ahead (cf. Tables 5 and 6).

When evaluating the performance gain from vectorization (cmp. Table 5), one thing is noticeable. For the Core2* systems, the speed-up is about 3.5×, but for the Nehalem and Dunnington it's only around 3× (cf. Table 5). Our explanation is that the Intel compiler does a pretty good job in auto-vectorizing and code optimization, resulting in a higher baseline. The next thing is that the speed-up from the TBB implementation to TBB+SIMD is noticeably lower than from baseline to SIMD. While the Core2* and Nehalem systems achieve only 2.6 to 2.8×, the speed-up is even less on the Dunnington system (1.9×). This is due to the bandwidth-bound nature of our algorithm. The higher number of threads consumes the total available memory bandwidth before the peak computation rate is reached. A similar result could be shown in previous experiments on GPUs (not yet published).

The results in Table 7 come up to the expectations of the Nehalem platform. Having only twice as much cores as the Core2Extreme the speed-up was between 2.63 for TBB+SIMD and 3.09× for OpenMP. One could argue that a good part of this effect is due to the Intel compiler. But with gcc the speed-up is 2.58 to 2.85× – still significantly higher than 2×. This is explained by the bandwidth-boundedness of the algorithm. The higher main memory bandwidth results in super-linear speed-ups.

As one can see in Table 4 the CUDA implementation suffers from its start-up overhead and is considerably slower than the Dunnington system for the volume size of 128^3 voxels. But it is already faster by a factor of 1.34× for 256^3 volumes and gains an even bigger advantage on 512^3 volumes where it outperforms the fastest CPU-based system by a factor of 3.38×.

Implementation	Core2Duo Speed-up	Core2Extreme Speed-up	Nehalem Speed-up	Dunnington Speed-up
single-threaded	1.02×	1.00×	1.20×	1.20×
SIMD	1.01×	1.00×	0.99×	1.00×
OpenMP	0.48×	1.00×	3.09×	6.15×
TBB	0.50×	1.00×	2.81×	5.66×
TBB+SIMD	0.48×	1.00×	2.63×	3.86×

Table 7. Comparing performance of different CPUs. Speed-up is relative to C2E and was normalized by clock rate.

5 Discussion

Using current libraries, multi-threading can be implemented easily if the problem has such a promotive nature as the backprojection algorithm. The bottom line for both multi-threading libraries investigated is that they scale nicely with the number of cores (when normalized by the clock speed).

Vectorization on the other hand is more difficult to implement. It is a great way to get more performance out of existing hardware. But compared to the speed-ups from multi-threading, the effort of vectorization looks not worthwhile. This will change in the future, when chips with wider SIMD units become available. Our experiences have shown that the step from 4-wide SIMD to 16-wide requires only minor modifications of the source code.

Different CPUs behave numerically equivalent. The results using SSE instructions differ only negligible from scalar computations. This is shown by the worst mean squared error (MSE) of all CPU implementations, which was only 0.001 (range: $0 \ldots 4095$). The CUDA implementation deviated from the reference implementation by about 8.1 HU^2. This error is still acceptable for clinical purposes. Note that since we reconstructed real acquired data we do not even know the true values. Thus we can not say if the GPU result is better or worse. But this example shows that the accuracy of different hardware platforms is an issue that should be further investigated. Accuracy measurements using simulated phantoms would be required to judge the image quality.

One reason that graphics cards are so fast is that they have fast on-board memory which offers high bandwidth. Its size is typically in the order of 1 or 2 GB. Volumes that do not fit in this amount of memory have to be split into several parts which are reconstructed independently. This is not only a cumbersome task but also involves transfers of huge amounts of data from and to the computer's main memory. If these transfers cannot be hidden using double buffering techniques they will decrease the reconstruction speed.

6 Conclusion and Outlook

In this paper we have compared five different approaches to exploiting the inherent parallelism of the backprojection algorithm. Each of them shows different potential for accelerating reconstruction algorithms. Their common denominator is that they are limited by the main memory bandwidth.

There exist further optimization methods which we did not investigate in this work. With loop unrolling and software pipelining the compiler and the developer can try to hide latencies of main memory accesses. Moreover, cache-aware implementations can take advantage of the large L2 (and L3) caches of the most recent CPUs. They can keep data close to the compute unit for a longer time and re-use it more often before it is written back to main memory.

The RabbitCT benchmark framework allows rapid development and assessment of different backprojection implementations. It is a suitable platform for prototyping and performance comparison of backprojection algorithms. We used it to evaluate maximum system performance only but more detailed measurements on scalability with only a small number of threads could be performed as well.

A variety of different hardware architectures exists to date. And the diversity will further increase. Future hardware generations (Intel's Larrabee, AMD's Fusion) will bring more heterogeneity. This makes it harder for programmers to exhaust their full power.

Developers are therefore supported by different parties. Modern compilers provide auto-vectorization options which are improved with every release. Software vendors develop tools that analyze code for parallelism and suggest threading strategies. Others provide new tools for debugging of multi-threaded programs. Besides the two libraries used in this work, more frameworks exist to ease multi-threaded programming. One example, Cilk++ [12], shows similarities to TBB. A special compiler transforms annotated sequential code into a parallel program. Sequoia [13] on the other hand strives to enable the programmer to create programs which are aware of the memory hierarchy and make the most of it. However it seems like this project is not under development any more.

Different libraries and novel platforms will be subject of our upcoming research. The recent results will be available online at RabbitCT [4]

Acknowledgements. This work was supported by the Regional Computing Center Erlangen (RRZE) and Intel Corporation who kindly provided the authors access to the pre-production systems.

References

1. Feldkamp, L., Davis, L., Kress, J.: Practical Cone-Beam Algorithm. Journal of the Optical Society of America **A1**(6) (1984) 612–619
2. Kunze, H., Härer, W., Stierstorfer, K.: Iterative extended field of view reconstruction. In Hsieh, J., Flynn, M.J., eds.: Medical Imaging 2007: Physics of Medical Imaging. Volume 6510 of Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series., San Diego (2007) 65105X
3. Rohkohl, C., Keck, B., Hofmann, H.G., Hornegger, J.: Technical note: Rabbitct—an open platform for benchmarking 3d cone-beam reconstruction algorithms. Medical Physics **36**(9) (2009) 3940–3944
4. Pattern Recognition Lab, Friedrich-Alexander University Erlangen: RabbitCT, Open Platform for Worldwide Comparison in Backprojection Performance (2009)
5. Intel Corp.: Threading Building Blocks (2009)
6. OpenMP: The OpenMP API specification for parallel programming. Website (2009) Available online at http://www.openmp.org/.
7. Seiler, L., Carmean, D., Sprangle, E., Forsyth, T., Abrash, M., Dubey, P., Junkins, S., Lake, A., Sugerman, J., Cavin, R., Espasa, R., Grochowski, E., Juan, T., Hanrahan, P.: Larrabee: A Many-Core x86 Architecture for Visual Computing. In: SIGGRAPH '08: ACM SIGGRAPH 2008 papers, Los Angeles, ACM (2008) 1–15
8. NVIDIA Corp.: NVIDIA CUDA Compute Unified Device Architecture Programming Guide (2007)
9. Faugeras, O.: Three-Dimensional Computer Vision (Artificial Intelligence). The MIT Press Cambridge (1993)
10. Wiesent, K., Barth, K., Navab, N., Durlak, P., Brunner, T., Schuetz, O., Seissler, W.: Enhanced 3-D-reconstruction algorithm for C-arm systems suitable for interventional procedures. IEEE Transactions on Medical Imaging **19**(5) (2000) 391–403
11. Scherl, H., Keck, B., Kowarschik, M., Hornegger, J.: Fast GPU-Based CT Reconstruction using the Common Unified Device Architecture (CUDA). In Frey, E.C., ed.: Nuclear Science Symposium, Medical Imaging Conference 2007. Volume 6., Honolulu (2007) 4464–4466

12. Frigo, M.: Multithreaded Programming in Cilk. In: Proceedings of the 2007 international workshop on Parallel symbolic computation, New York, NY, USA, ACM (2007) 13–14
13. Fatahalian, K., Knight, T.J., Houston, M., Erez, M., Horn, D.R., Leem, L., Park, J.Y., Ren, M., Aiken, A., Dally, W.J., Hanrahan, P.: Sequoia: Programming the memory hierarchy. In: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing. (2006)