Pushing the limits for medical image reconstruction on recent standard multicore processors

Jan Treibig\textsuperscript{1}, Georg Hager\textsuperscript{1}, Hannes G. Hofmann\textsuperscript{2}, Joachim Hornegger\textsuperscript{2} and Gerhard Wellein\textsuperscript{1}

Abstract
Volume reconstruction by backprojection is the computational bottleneck in many interventional clinical computed tomography (CT) applications. Today vendors in this field replace special purpose hardware accelerators with standard hardware such as multicore chips and GPGPUs. Medical imaging algorithms are on the verge of employing high-performance computing (HPC) technology, and are therefore an interesting new candidate for optimization. This paper presents low-level optimizations for the backprojection algorithm, guided by a thorough performance analysis on four generations of Intel multicore processors (Harpertown, Westmere, Westmere EX, and Sandy Bridge). We choose the RABBIT CT benchmark, a standardized testcase well supported in industry, to ensure transparent and comparable results. Our aim is to provide not only the fastest possible implementation but also compare with performance models and hardware counter data in order to fully understand the results. We separate the influence of algorithmic optimizations, parallelization, SIMD vectorization, and microarchitectural issues and pinpoint problems with current SIMD instruction set extensions on standard CPUs (SSE, AVX). The use of assembly language is mandatory for best performance. Finally, we compare our results to the best GPGPU implementations available for this open competition benchmark.

Keywords
performance optimization, medical imaging, performance analysis, SIMD, SandyBridge (AVX)

1 Introduction and related work
1.1 Computed tomography
Computed tomography (CT) (Kak and Slaney 2001) is nowadays an established technology to non-invasively determine a three-dimensional (3D) structure from a series of projections of an object. Beyond its classic application area of static analysis in clinical environments the use of CT has accelerated substantially in recent years, e.g. towards material science or time-resolved scans supporting interventional cardiology. The numerical volume reconstruction scheme is a key component of modern CT systems and is known to be very compute-intensive. Acceleration through special-purpose hardware such as field programmable gate arrays (FPGAs) (Heigl and Kowarschik 2007) is a typical approach to meet the constraints of real-time processing. Integrating non-standard hardware into commercial CT systems adds considerable costs both in terms of hardware and software development, as well as system complexity. From an economic view the use of standard x86 processors would thus be preferable. Driven by Moore’s law the compute capabilities of standard CPUs have now the potential to meet the requested CT time constraints.

The volume reconstruction step for recent C-arm systems with flat panel detector can be considered as a prototype for modern clinical CT systems. Interventional C-arm CTs, such as the one sketched in Figure 1, perform the rotational acquisition of 496 high-resolution X-ray projection images (1248 \times 960 pixels) in 20 seconds (Strobel et al. 2009). This acquisition phase sets a constraint for the maximum reconstruction time to attain real-time reconstruction. In practice, filtered backprojection (FBP) methods such as the Feldkamp algorithm (Feldkamp et al. 1984) are widely used for performance reasons. The algorithm consists of 2D pre-processing steps, backprojection, and 3D

\textsuperscript{1} Erlangen Regional Computing Center, Erlangen, Germany
\textsuperscript{2} Pattern Recognition Lab, University Erlangen-Nuremberg, Germany

Corresponding author:
Jan Treibig, Erlangen Regional Computing Center Martensstrasse 1, 91058 Erlangen, Germany.
Email: jan.treibig@rrze.uni-erlangen.de
post-processing. Volume data is reconstructed in the back-projection step, making it by far the most time-consuming part (Heigl and Kowarschik 2007). It is characterized by high computational intensity, non-trivial data dependencies, and complex numerical evaluations but also offers an inherent embarrassingly parallel structure. In recent years hardware-specific optimization of the Feldkamp algorithm has focused on GPUs (Mueller and Yagel 1998; Mueller et al. 2007; Scherl et al. 2007a; Okitsu et al. 2010; Papenhausen et al. 2011) and IBM Cell processors (Kachelrieß et al. 2007; Scherl et al. 2007b). For GPUs in particular, large performance gains compared with CPUs were reported (Mueller et al. 2007) or documented by the standardized RABBIT CT benchmark (Rohkohl et al. 2009a), e.g. for cardiac imaging. However, both approaches incorporate several backprojection steps, making performance improvements on the RABBITCT benchmark valuable for them as well. The same holds for industrial CTs in material science used in non-destructive testing (NDT), which additionally run at higher resolution and thus further increase the computational requirements (Keck et al. 2009). The high computational demand of backprojection algorithms together with the growing acceptance of parallel computing in medical applications make them interesting new candidates for the interface to high-performance computing (HPC).

1.2 Modern processors

The steadily growing transistor budget is still pushing forward the compute capabilities of commodity processors at rather constant price and clock speed. Improvement and scaling of established architectural features in the core (such as, e.g., single instruction multiple data [SIMD] and simultaneous multi-threading [SMT]; see below for details) in addition to increasing the number of cores per chip lead to peak performance levels that enable standard CPUs to meet the time constraints of interventional CT systems. Optimized single-core implementations are thus mandatory for reconstruction algorithms. Complemented by standard optimization techniques, a highly efficient SIMD code is the key performance component. Owing to the non-trivial data parallelism in the reconstruction scheme, SIMD optimizations need to be done at a low level, i.e. down to assembly language. However, these efforts will pay off for future computer architectures since SIMD vector lengths are expected to increase considerably in the future; hence, efficient SIMD implementations will be required to further benefit from increasing peak performance.

Scaling SIMD width is a straightforward way to increase raw core performance and optimize energy efficiency (i.e. maximize the flop /Watt ratio), which is known to be a critical aspect of future HPC systems. Recently Intel has doubled the SIMD width from the SSE instruction set.
The International Journal of High Performance Computing Applications 27(2)

Table 1. Test machine specifications. The cacheline size is 64 bytes for all processors and cache levels. The update benchmark results were obtained with the likwid-bench tool.

| Model                      | Intel Harpertown | Intel Westmere | Intel Westmere EX | Intel Sandy Bridge |
|----------------------------|------------------|----------------|-------------------|--------------------|
| Label                      | Xeon X5482       | Xeon X5670     | Xeon E7-4870      | Core i7-2600       |
| Clock (GHz)                | HPT              | WEM            | WEX               | SNB                |
| Node sockets/cores/threads | 3.2              | 2.66           | 2.40              | 3.4 (3.5 turbo)    |
| Socket L1/L2/L3 cache      | 2/8/-            | 2/12/24        | 4/40/80           | 1/4/8              |
| Bandwidths [GB/s]:         |                  |                |                   |                    |
| Theoretical socket BW      | 12.8             | 32.0           | 34.2              | 21.3               |
| Update (1 thread)          | 5.9              | 15.2           | 8.3               | 16.5               |
| Update (socket)            | 6.2              | 20.3           | 24.6              | 17.3               |
| Update (node)              | 8.4              | 39.1           | 98.7              | –                  |

(128-bit registers) to AVX (Intel 2011a) (256-bit registers, with larger widths planned for future designs). The latter is implemented in the ‘Sandy Bridge’ architecture. Other ongoing projects are pointing in the same direction, such as Intel’s Many Integrated Core (MIC)2 architecture or the Chinese Godson-3B chip.3 Wider SIMD units do not change the core’s complexity substantially since the optimal instruction throughput does not depend on the SIMD width. However, the benefit in the application strongly depends on the level and the structure of data parallelism as well as the capability of the compiler to detect and exploit it. Compilers are very effective for simple code patterns such as streaming kernels,4 while more complex loop structures require manual intervention, at least to the level of compiler intrinsics. This is the only safe way to exploit the full potential of SIMD capabilities. The impact of wider SIMD units on programming style is still unclear since this trend is currently starting to accelerate. Of course, wide SIMD execution is most efficient for in-cache codes because of the large ‘DRAM gap’.

SMT is another technology to improve core utilization at low architectural impact and energy cost. SMT should be most beneficial for those ‘in-cache codes’ that have limited single thread efficiency due to bubbles in arithmetic/logic pipelines, and where cache bandwidth is not the dominating factor. Naively one would not expect improvements from SMT if a code is fully SIMD-vectorized since SIMD is typically applied to simple data-parallel structures, which often lead to efficient pipelining. Since many programmers do not care about pipeline occupancy in their application the benefit of SMT is often tested in an experimental way, without a true understanding of why it does or does not improve performance.

This paper is organized as follows. In Section 3 we present the theoretical background and perform a first analysis of the backprojection algorithm implemented in the RABBIT framework using simple metrics such as arithmetic throughput and memory bandwidth as guidelines for estimating performance. We address processors from four generations of Intel’s x86 family (Harpertown, Westmere, Westmere EX, and Sandy Bridge). Basic optimization rules such as minimizing overall work are applied. In Section 4 we show how to vectorize the inner loop kernel using SSE and AVX instructions, and discuss the possible benefit of multithreading with SMT. Section 5 provides an in-depth performance analysis, which will show that simple bandwidth or arithmetic throughput models are inadequate to estimate the performance of the algorithm. OpenMP parallelization and related optimizations such as ccNUMA placement and bandwidth reductions are discussed in Section 6. Performance results for cores, sockets, and nodes on all four platforms are given in Section 7, where we also interpret the effect of the different optimizations discussed earlier and validate our performance model. Finally we compare our results with current GPGPU implementations in Section 8.
This benchmark reflects the data streaming properties of the reconstruction algorithm and is thus better suited than STREAM as a baseline for a quantitative performance model.

We use the Intel C/C++ compiler in version 12.0; since most of our performance-critical code is written in assembly language, this choice is marginal, however. Thread affinity, hardware performance monitoring, and low-level benchmarking were implemented via the LIKWID tool suite (Treibig et al. 2010), using the tools likwid-pin, likwid-perfctr, and likwid-bench, respectively.

3 The algorithm

3.1 Theory

The RABBIT-CT dataset consists of \( N = 496 \) projection images \( I_p \) acquired by a C-arm system. The projections are already pre-processed and filtered. Hence, only the backprojection step is considered in the present work. Each projection image is accompanied by a projection matrix \( A_n \in R^{3 \times 4} \) (Navab et al. 1998; Wiesent et al. 2000). It encodes the complete projection geometry, including reproducible deviations from the ideal Feldkamp geometry (Wiesent et al. 2000). Using \( A_n \), the perspective projection of an arbitrary point \( x = (x, y, z)^T \) in 3D space onto the point \( p \) in the \( u-v \) image plane of the \( n \)th view can be expressed as (Hartley and Zissermann 2004)

\[
\tilde{p}_n = A_n \hat{x},
\]

\( \hat{x} = (x^T, 1)^T \) and \( \tilde{p} = (u, v, 1)^T w \). Note that the equality is in homogeneous coordinates and therefore up to scale. For convenience, we further define

\[
u_n(x) = \frac{w_n(x)}{w_n(x)},
\]

\[
v_n(x) = \frac{\tilde{p}_{n1}}{w_n(x)}, \text{ and}
\]

\[
w_n(x) = \tilde{p}_{n2}.
\]

The \( N \) filtered projection images \( I_p \) are backprojected into the volume \( F \). The value of a voxel at position \( x = (x, y, z) \) is determined as

\[
F(x) = \sum_{n=1}^{N} \frac{1}{w_n(x)} I_p(u_n(x), v_n(x)).
\]

Since Equation (1) is only defined up to scale, \( A_n \) can be constructed such that the scaling factor \( \tilde{p}_{n2} \) corresponds to the distance weight \( w \) in the backprojection formula (5) (Wiesent et al. 2000).

Note that in practice one deals with image data that has a finite pixel resolution. The projection of a voxel will in general not hit one pixel of the 2D CT image exactly. Therefore, the projection value is computed by bilinear interpolation of the four closest pixels.

3.2 Code analysis

The basic backprojection algorithm (as provided by the RABBIT-CT framework, see Listing 1) is usually implemented in single precision (SP) and exhibits a streaming access pattern for most of its data traffic. One volume reconstruction uses 496 CT images (denoted by \( I \)) of 1248 x 960 pixels each (ISX x ISY). The volume size is 256^3 mm^3. MM is the voxel size and changes depending on the number of voxels. The most common resolution in present clinical applications is 512 voxels in each direction (denoted by the problem size \( L \)). The algorithm computes the contributions to each voxel across all projection images, and the reconstructed volume is stored in array \( VOL \). Voxel coordinates (indices) are denoted by \( x, y, \) and \( z \), while pixel coordinates are called \( u \) and \( v \). See Figure 3 for the geometric setup.

**Listing 1:** Voxel update loop nest for the plain backprojection algorithm. This gets executed for each projection \( I \). All variables are of type float unless indicated otherwise. The division into parts (see the text) is only approximate since there is no one-to-one correspondence to the SIMD-vectorized code.

```c
wz = offset_z;
for(int z = 0; z < L; z++)
    wz += MM);
    wy = offset_y;
for (int y = 0; y < L; y++)
    wy += MM) {
    wx = offset_x;
    valt1 = 0.0f; valtr = 0.0f;
    valbl = 0.0f; valbr = 0.0f;
    // Part 1 ---------------------------
    for (int x = 0; x < L; x++)
        wx += MM) {
            uw = (A[0]*wx+A[3]*wy+A[6]*wz+A[9]);
            vw = (A[1]*wx+A[4]*wy+A[7]*wz+A[10]);
            w = (A[2]*wx+A[5]*wy+A[8]*wz+A[11]);
            u = uw * 1.0f/w; v = vw * 1.0f/w;
            int iu = (int)u; iv = (int)v;
            scalu = u - (float) iu;
            scalv = v - (float) iv;
            // Part 2 ---------------------------
            if (iv > 0 && iv<ISY)
                if (iu > 0 && iu<ISX)
                    valt1 = (i)iv*ISX + iu);
                    if (iu > -1 && iu<ISX-1)
                        valtr = (i)iv*ISX + iu+1];
                } if (iv > -1 && iv < ISY-1) {
                    if (iu > 0 && iu<ISX)
                        valbl = (i)iv+1)*ISX + iu];
                        if (iu > -1 && iu<ISX-1)
                            valbr = (i)iv+1)*ISX + iu+1];
                } // Part 3 ---------------------------
                vall = scalv*valbl + (1.0f-sclav)*valt1;
                valr = scalv*valbr + (1.0f-sclav)*valtr;
                fx = scalu*valr + (1.0f-scalu)*vall;
                VOL [z*L*L + y*L + x] += 1.0f/(w*w)*fx;
        } // x
    } // y
} // z
```
The aggregate size of all projection images is \( \approx 2.4 \) GB. One voxel sweep incurs a data transfer volume which consists of the loads from the projection image and an update operation \( \text{VOL}[i] += s \), see in Listing 1) to the voxel array. The latter causes 8 bytes of traffic per voxel and results (for problem size \( 512^3 \)) in a data volume of 1 GB, or 496 GB for all projections. The traffic caused by the projection images is not easy to quantify since it is not a simple stream; it is defined by a ‘beam’ of locations slowly moving over the projection pixels as the voxel update loop nest progresses. It exhibits some temporal locality since neighboring voxels are projected on proximate pixels of the image, but there may also be multiple streams with large strides. On the computational side, the basic version of this algorithm performs 13 additions, 5 subtractions, 17 multiplications, and 3 divides.

### 3.3 Simple performance models

Based on this knowledge about data transfers and arithmetic operations we can derive a first rough estimate of expected upper performance bounds. The arithmetic limitation results in 21 cycles per vectorized update (4 and 8 inner loop iterations for SSE and AVX, respectively), assuming full vectorization and a throughput of one divide per cycle. This takes into account that all architectures under consideration can execute one addition and one multiplication per cycle, neglects the slight imbalance of additions versus multiplications, and assumes that the pipelined repps instruction can be employed for the divisions (see Section 4.1 for details).

On the other hand, runtime can also be estimated based on the data transfer volume and the maximum data transfer capabilities of the nodes measured with the synthetic update benchmark described in Section 2. Table 2 shows upper performance bounds for a full \( 512^3 \) reconstruction based on arithmetic and bandwidth limitations on the four systems in the testbed. Performance is given in billions of voxel updates per second (GUP/s),\(^5\) where one ‘update’ represents the reconstruction step of one voxel using a single image. The large expected performance for the single socket (quad-core) Sandy Bridge under arithmetic limitation is caused by its wide AVX vector size and its faster clock. While above predictions seem to indicate a strongly memory-bound situation, they are far from accurate: the runtime is governed by the number

| Limitation       | HPT | WEM | WEX | SNB |
|------------------|-----|-----|-----|-----|
| Arithmetic (GUP/s) | 4.86 | 6.75 | 13.9 | 5.31 |
| Bandwidth (GUP/s) | 1.06 | 4.90 | 11.1 | 2.15 |

---

**Table 2.** Analytic upper performance bounds determined by either algorithmic or bandwidth properties of the high level algorithm. (full nodes; see Table 9 for label definitions).

---

**Figure 3.** Setup geometry for generating the CT projection images. The size of the volume is always \( 256^3 \) mm\(^3\), but the number of voxels may vary. The \( x\)-\( y\)-\( z\) space represents the volume while the \( u\)-\( v\) plane represents a filtered projection.
of instructions and the cycles it takes to execute these instructions. A reduction to the purely ‘useful’ work, i.e., to arithmetic operations, can not be made since this algorithm is non-trivial to vectorize due to the scattered load of the projection image; it therefore involves many more non-arithmetic instructions (see Section 4.1 for details). We show later that a more careful analysis leads to a completely different picture, and that further optimizations can change the bottleneck analysis considerably.

In order to have a better view on low-level optimizations we divide the algorithm into three parts:

1. Geometry computation: Calculate the index of the projection of a voxel in pixel coordinates.
2. Load four corner pixel values from the projection image.
3. Interpolate linearly for the update of the voxel data.

### 3.4 Algorithmic optimizations

The first optimizations for a given algorithm must be on a hardware-independent level. Beyond elementary steps such as moving invariant computations out of the inner loop body and reducing the divides to one reciprocal (thereby reducing the flop count to 31), a main optimization is to minimize the workload. Voxels located at the corners and edges of the volume are not visible on every projection, and can thus be ‘clipped off’ and skipped in the inner loop. This is not a new idea, but the approach presented here improves the work reduction from 24% (Hofmann et al. 2010) to nearly 39%.

The basic building block for all further steps is the update of a consecutive line of voxels in x direction, covered by the inner loop level in Listing 1. We refer to this as the ‘line update kernel’. The geometry, i.e., the position of the first and the last relevant voxel for each projection image and line of voxels is precomputed. This information is specific for a given geometric setup, so it can be stored and used later during the backprojection loop. Reading the data from memory incurs an additional transfer volume of $512^2 \times 496 \times 4$ bytes = 496 MBytes (assuming 16-bit indexing), which is negligible compared with the other traffic. The advantage of line-wise clipping is that the shape of the clipped voxel volume is much more accurately tracked than with the blocking approach described in Hofmann et al. (2010).

The conditionals (lines 23 and 30 in Listing 1), which ensure correct access to the projection image, involve no measurable overhead for the scalar case due to the hardware branch prediction. However, for vectorized code they are potentially costly since an appropriate mask must be constructed whenever there is the possibility that a SIMD vector instruction accesses data outside the projection (Hofmann et al. 2010). To remove this complication, separate buffers are used to hold suitably zero-padded copies of the projection images, so that there is no need for vector masks. The additional overhead is far outweighed by the performance advantage for fully vectorized code execution. The conditionals are also effectively removed by the clipping optimization described above, but we need a code version without clipping for validating our performance model later.

Note that a similar effect could be achieved by peeling off scalar loop iterations to make the length of the inner loop body a multiple of the SIMD vector size and ensure aligned memory access. However, this may introduce a significant scalar component especially for small problem sizes and large vector lengths.

### 4 Single-core optimizations

For all further optimizations we choose an implementation of the line update kernel in C as the baseline. All algorithmic optimizations from Section 3.4 have already been applied.

The performance of present processors on the core level relies on instruction-level parallelism (ILP) by pipelined and superscalar execution, and data-parallel operations (SIMD). We also regard SMT as a single-core optimization since it is a hardware feature to increase the efficiency of the execution units by filling pipeline bubbles with useful work: The idea is to duplicate parts of the hardware resources (control logic, registers) in order to allow a quasi-simultaneous execution of different threads while sharing other parts such as, e.g., floating-point pipelines. In a sense this is an alternative to outer loop unrolling, which can also provide multiple independent instruction streams.

We elaborate on the SIMD vectorization here and comment on the benefit of SMT in Section 5.2.

#### 4.1 SIMD vectorization

No current compiler is able to efficiently vectorize the backprojection algorithm, so we have implemented the code directly in x86 assembly language. Using SIMD intrinsics could ease the vectorization but adds some uncertainties with regard to register scheduling and hence does not allow full control over the instruction code. All data is aligned to enable packed and aligned loads/stores of vector registers (16 (SSE) or 32 (AVX) bytes with one instruction).

The line update kernel operates on consecutive voxels. For part 1 of the algorithm classical vectorization with working on multiple voxels at the same time can be employed (see Section 3.3). This part is arithmetically limited and fully benefits from the increased register width. The division is replaced by a reciprocal. SSE provides the fully pipelined rcpps instruction for an approximate reciprocal with reduced accuracy compared with a full divide. This approximation is sufficient for this algorithm, and results in an accuracy similar to GPGPU implementations. An analysis of the impact of the approximate reciprocal on performance and accuracy is presented in Section 7.2. The integer cast (line 18) is implemented via the vectorized
hardware rounding instruction roundps, which was introduced with SSE4.

Part 2 of the algorithm cannot be directly vectorized. The projection value is computed by bilinear interpolation of the four closest pixels (top left (valtl), top right (valtr), and bottom left (valbl), bottom right (valbr)). As illustrated in Figure 5, valtl and valtr as well as valbl and valbr can be loaded pairwise. Moreover, the classic vectorization approach of part 1, operating on multiple voxels at the same time, cannot be retained here since neighboring voxels will in general not be projected onto consecutive pixels. Below we describe how we optimized the loads and interpolation in \( v \) direction. Figure 4 shows the steps involved in vectorizing part 2 and the first linear interpolation in more detail. For the sake of simplicity, we consider a vector of four voxels with indices 0–3, but the scheme can be easily extended to wider vectors.

Since the pixel coordinates from step 1 are already in a vector register, the index calculation for, e.g., \( iv*ISX+iu \) and \( (iv+1)*ISX+iu \) (lines 25, 27, 32, and 34 in Listing 1) uses packed SIMD instructions. We compute the first part of the interpolation for voxels 0 and 2 simultaneously, and then for voxels 1 and 3. Therefore, we duplicate the weighting vector scalv such that one copy contains the weights for voxels 0 and 2 (twice), and the other one for voxels 1 and 3 (step 1 in Figure 4). With one load at the index of valtl we implicitly load valtr into the second vector element. Again we create two vectors, one containing the top left and right pixel values for voxels 0 and 2, and one for 1 and 3 (step 2 in Figure 4). The same is done for the bottom values. The resulting vectors are called valt and valb, respectively, in Figure 4. Note that the cost for this construct increases with wider SIMD registers because two load operations are required per voxel. Next, we multiply those vectors with the corresponding scalv vector in step 3 of Figure 4 and get vectors with the interpolation result in the \( v \) direction. The elements are still alternating left and right, and for voxels 0 and 2, or 1 and 3. Therefore, in step 4, we reorder the elements to obtain a vector containing only the left values for all voxels and one containing the right values. This scheme is implemented using the SSE3 instructions movsdup and movshdup for duplication of the scalv values. The final reordering to enable classic vectorization in part 3 of the algorithm also uses those instructions, together with blendps, which interleaves the values to bring them into the correct order again. The conversion of the index into a general-purpose register, which is needed for addressing the load of the data and the scattered pairwise loads, is costly in terms of necessary instructions. Moreover, the runtime increases linearly with the width of the registers due to the pairwise loads. This implies that the whole operation is limited by instruction throughput.

We consider two SSE implementations, which only differ in part 2 of the algorithm. Version 1 (V1) converts the floating point values in the vector registers to four quadwords and stores the result back to memory (cache, actually). Single index values are then loaded to general-purpose registers one by one. Version 2 (V2) does not store to memory but instead shifts all values in turn to the lowest position in the SSE register, from where they are moved directly to a general-purpose register using the cvtss2si instruction.

The remainder of part 3, the second part (\( u \) direction) of the bilinear interpolation and the voxel update, is again trivially vectorizable and fully benefits from wider SIMD registers.

Note that any further inner loop unrolling beyond what is required by SIMD vectorization would not show any benefit due to register shortage; however, as will be shown later, SMT can be used to achieve a similar effect.
Table 3. Intel Architecture Code Analyzer (IACA) (Intel 2011b) analysis for Intel Westmere.

| Variant | 0 | 1 | 2 | 3 | 4 | 5 | TP | μOPs | CP |
|---------|---|---|---|---|---|---|----|------|----|
| V1      | 15| 21| 24| 3 | 3 | 19| 24 | 85   | 54 |
| V2      | 20| 27| 16| 1 | 1 | 20| 27 | 85   | 71 |

4.2 AVX implementation

In theory, the new AVX instruction set extension doubles the peak performance per core as compared with SSE. The backprojection cannot fully benefit from this advantage because the number of required instructions increases linearly with the register width in part 2 of the algorithm. For arbitrary SIMD vector lengths a hardware gather operation would be required to prevent this part from becoming a severe bottleneck.

Also the limited number of AVX instructions that natively operate on 256-bit registers prohibits more sophisticated variants of part 2; only the simple version V1 could be implemented. A register-to-register variant would be possible only at the price of a much larger instruction count, so this alternative was not considered. Despite these shortcomings, an improvement of 25% could be achieved with the AVX kernel on Sandy Bridge (see Section 7 for detailed performance results).

AVX2, which will be implemented in the Intel Haswell processor in 2012, will fix some issues by introducing a hardware gather instruction and a more complete instruction set operating on the full SIMD register width.

5 In-depth performance analysis

5.1 Analytic performance model

Popular performance models for bandwidth-limited algorithms reduce the influence on the runtime to the algorithmic balance, taking into account the sustained main memory performance (Hager and Wellein 2010). This assumption works well in many cases, especially when there is a large gap between arithmetic capabilities and memory bandwidth. Recently it was shown (Treibig and Hager 2009) that this simplification is problematic on newer architectures such as Intel Core i7 because of their exceptionally large memory bandwidth per socket; e.g. a single Nehalem core cannot saturate the memory interface (Treibig et al. 2009). Moreover the additional L3 cache decouples core instruction execution from main memory transfers and generally provides a better opportunity to overlap data traffic between different levels of the memory hierarchy. Note that in a multithreaded scenario the simple bandwidth model can indeed work as long as multiple cores are able to saturate the socket bandwidth. This property depends crucially on the algorithm, of course.

In Treibig and Hager (2009) we have introduced a more detailed analytic method for cases where the balance model is not sufficient to predict performance with the required accuracy, and the in-cache data transfers account for a significant fraction of overall runtime. This model is based on an instruction analysis of the innermost loop body and runtime contributions of cacheline transfer volumes through the whole memory hierarchy. We first cover single-threaded execution only and then generalize to multithreading on the socket once the basic performance limitations are understood.

A useful starting point for all further analysis is the single-thread runtime spent executing instructions with data loaded from L1 cache. The Intel Architecture Code Analyzer (IACA) (Intel 2011b) was used to analytically determine the runtime of the loop body. This tool calculates the raw throughput according to the architectural properties of the processor under the assumption that all data resides in the L1 cache. It supports Westmere and Sandy Bridge (including AVX) as target architectures. The results for Westmere are shown in Table 3 for the two SSE kernel variants described above (all entries except μOPs are in core cycles). Execution times are calculated separately for all six issue ports (0... 5). (A μOP is a RISC-like 'micro-instruction'; x86 processors perform an on-the-fly translation of machine instructions to μOPs, which are the ‘real’ instructions that get executed by the core.) Apart from the raw throughput (TP) and the total number of μOPs the tool also reports a runtime prediction taking into account latencies on the critical path (CP). Based on this prediction V1 should be faster than V2 on Westmere. However, the measurements in Table 3 show the opposite result. The high pressure on the load issue port (2) together with an overall high pressure on all ALU issue ports (0, 1, and 5) seems to be decisive. In V2 the pressure on port 2 is much lower, although the overall pressure on all issue ports is slightly larger.

In Table 4 we report the results for the Sandy Bridge architecture with SSE and AVX. The pressure on the ALU ports is similar, but due to the doubled SSE load performance Sandy Bridge needs only half the cycles for the loads in kernel V1. V1 is therefore faster than V2 on Sandy Bridge (see Table 5).

So far we have assumed that all data resides in the L1 cache. The data transfers required to bring cachelines into L1 and back to memory are modeled separately. We assume that there is no overlap between data transfers and instruction execution. This is true at least for the L1 cache: it can either communicate with L2 to load or evict a cacheline, or it can deliver data to the registers, but not both at the same time. As a first approximation we also (pessimistically)
assume that this ‘no-overlapping’ condition holds for all caches, and that a data transfer between any two adjacent levels in the memory hierarchy does not overlap with anything else. Since the smallest transfer unit is a 64-byte cacheline, the analysis will from now on be based on a full ‘cacheline update’ (16 four-byte voxels), which corresponds to four (two) inner loop iterations when using SSE (AVX).

We only consider the data traffic for voxel updates; the image data traffic is negligible in comparison, hence we assume that all image data comes from L1 cache. It takes two cycles to transfer one cacheline between adjacent cache levels over the 256-bit unidirectional data path. Every modified line must eventually be evicted, which takes another two cycles. Figure 6 shows a full analysis, in which the core execution time for a complete cacheline update is based on the measured cycles from Table 5. On the three architectures with L3 cache the simplification is made that the ‘Uncore’ part (L3 cache, memory interface, and QuickPath interconnect) runs at the same frequency as the core, which is not strictly true but does not change the results significantly. It was shown for the Nehalem-based architectures (Westmere and Westmere EX) that they can overlap instruction execution with reloading data from memory to the last level cache (Treibig et al. 2009). Hence, the model predicts that the in-core execution time is much larger than all other contributions, which makes this algorithm limited by instruction throughput for single-core execution. On Sandy Bridge, the AVX kernel requires 76.2 cycles for one vectorized loop iteration (eight updates). This results in 152 cycles instead of 178 cycles (SSE) for one cacheline update.

Based on the runtime of the loop kernel we can now estimate the total required memory bandwidth for multi-threaded execution if all cores on a socket are utilized, and also derive the expected performance (we consider the full volume without clipping), as shown in Table 6. We conclude that the multithreaded code is bandwidth-limited only on Harpertown, since the required socket bandwidth is above the practical limit given by the update benchmark (see Table 1). All other architectures are below their data transfer capabilities for this operation and should show no benefit from further bandwidth-reducing optimizations (see Section 6.2).

### 5.2 ILP optimization and SMT

At this point the analysis still neglects the possible benefit from SMT. SMT can improve the pipeline utilization for codes that suffer from dependencies, long-latency loads, instruction scheduling issues, or resource contention. At the same time it is important to understand that there can be no benefit if all threads running on the same core compete for a shared resource such as, e.g., a request queue.

As shown in the previous section, our implementation of the backprojection algorithm exhibits a strong discrepancy between the IACA ‘throughput’ and ‘critical path’ predictions. Owing to the complex loop body, register dependencies are unavoidable, resulting in many pipeline bubbles. Outer loop unroll and jam (interleaving two outer loop iterations in the inner body) is out of the question due to register shortage, but SMT can do a similar job and provide independent instruction streams using independent register sets. Since there is no shared resource apart from the core pipelines, running two threads on the two virtual cores of each physical core is expected to reduce the cycles taken for the cacheline update. However, the effect of using SMT is difficult to estimate quantitatively. See Section 7 below for complete parallel results.

### 6 OpenMP parallelization

OpenMP parallelization of the algorithm is straightforward and works with all optimizations discussed so far. For the thread counts and problem sizes under consideration here it is sufficient to parallelize the loop that iterates over all voxel volume slices (loop variable z in Listing 1). However, due to the clipped-off voxels at the edges and corners of the volume, simple static loop scheduling with default chunk size leads to a strong load imbalance. This can be easily corrected by using block-cyclic scheduling with a small chunk size (e.g. static,1).

Images are produced one by one during the C-arm rotation, and could at best be delivered to the application in batches. Since the reconstruction should start as soon as
images become available, a parallelization across images was not considered.

As shown in Section 5, the socket-level performance analysis does not predict strong benefits from bandwidth-reducing optimizations except on the Harpertown platform. However, since one can expect to see more bandwidth-starved processor designs with a more unbalanced ratio of peak performance to memory bandwidth in the future, we still consider bandwidth optimizations important for this algorithm. Furthermore, ccNUMA architectures have become omnipresent even in the commodity market, making locality and bandwidth awareness mandatory. In the following sections we describe a proper ccNUMA page placement strategy for voxel and image data, and a blocking optimization for bandwidth reduction. The reason why we present those optimizations in the context of shared-memory parallelization is that they become relevant only in the parallel case, since bandwidth is not a problem on all architectures for serial execution (see Section 5.1).

### 6.1 ccNUMA placement

The reconstruction algorithm uses essentially two relevant data structures: the voxel array and the image data arrays. Upon voxel initialization one can easily employ first-touch initialization, using the same OpenMP loop schedule (i.e. access pattern) as in the main program loop. This way each thread has local access (i.e. within its own ccNUMA domain) to its assigned voxel layers, and the full aggregate bandwidth of a ccNUMA node can be utilized.

Although the access to the projection image data is much less bandwidth-intensive than the memory traffic incurred by the voxel updates, ccNUMA page placement was implemented here as well. As mentioned in Section 3.4, the padded projection buffers are explicitly allocated and initialized in each locality domain, and a local copy is shared by all threads within a domain. Since the additional overhead for the duplication is negligible, this ensures conflict-free local access to all image data. The time taken to copy the images to the local buffers is included in the runtime measurements.

### 6.2 Blocking/unrolling

In order to reduce the pressure on the memory interface we use a simple blocking scheme for the outer loop over all images: projections are loaded and copied to the padded...
projection buffers in small chunks, i.e. \( b \) images at a time. The line update kernel (see Section 4) for a certain pair of \((y, z)\) coordinates is then executed \( b \) times, once for each projection. This corresponds to a \( b \)-way unrolling of the image loop and a subsequent jam into the next-to-innermost voxel loop (across the \( y \) voxel coordinate). At the problem sizes studied here, all of the voxel data for this line can be kept in the L1 cache and reused \( b - 1 \) times. Hence, the complete volume is only updated in memory \( 496/b \) instead of 496 times. Relatively small unrolling factors between 2 and 8 are thus sufficient to reduce the bandwidth requirements to uncritical levels even on ‘starved’ processors such as the Intel Harpertown.

This optimization is so effective that it renders proper ccNUMA placement all but obsolete; we will thus not report the benefit of ccNUMA placement in our performance results, although it is certainly performed in the code.

7 Results

In order to evaluate the benefit of our optimizations we have benchmarked different code versions with the 512³ case on all test machines. RAnrCT includes a benchmarking application, which takes care of timing and error checking. It reports total runtime in seconds for the complete backprojection. We performed additional hardware performance counter measurements using the likwid-perfctr tool. likwid-perfctr can produce high-resolution timelines of counter data and useful derived metrics on the core and node level without changes to the source code. Unless stated otherwise we always report results using two SMT threads per core. For all architectures apart from Sandy Bridge the line update kernel version V2 was used. On Sandy Bridge results for the SSE kernel V1 as well as for the AVX port of the V1 kernel are presented.

7.1 Validation of analytical predictions

To validate the predicted performance of the analytic model (see Section 5), single-socket runs were performed without the clipping optimization and SMT. Blocking was used on the Harpertown platform only, to ensure that execution is not dominated by memory access. Table 7 shows the measured performance and the deviation against the model prediction. This demonstrates that the model has a reasonable predictive power. It has been confirmed that the contribution of data transfers indeed vanishes against the core runtime, despite the fact that the total transfer volume is high and a first rough estimate based on data transfers and arithmetic throughput alone (Section 3) predicted a bandwidth limitation of this algorithm on all machines.

As a general rule, the IACA tool can provide a rough estimate of the innermost loop kernel runtime via static code analysis. Still it is necessary to further enhance the machine model to improve the accuracy of the predictions. In particular, the ability of the out-of-order scheduler to exploit superscalar execution was overestimated and has led to qualitatively wrong predictions.

Note that this example is an extreme case with all data transfers vanishing against core runtime. However, the approach also works for bandwidth-limited codes, as was shown by Treibig and Hager (2009).

7.2 Impact of optimizations on accuracy

One of the costly operations in this algorithm is the division involved. A possible optimization is to use the fully pipelined reciprocal (rcpps), which provides an approximation with 12-bit accuracy compared with the 24-bit accuracy of the regular divide instruction. The accuracy of the reciprocal can be improved to at least 21 bits using a Newton–Raphson iteration (Intel 1999). This requires four additional arithmetic instructions in the implementation. Table 8 shows the root mean square error (RMSE), the peak signal-to-noise ratio (PSNR), and the performance of three code alternatives (regular divide instruction, reciprocal, and reciprocal with Newton–Raphson iteration), on the Westmere test machine for the fully optimized case (full node). As usual in image processing, the PSNR was determined as the logarithm of a mean quadratic deviation from a reference image:

\[
PSNR = 10 \cdot \log_{10} \left( \frac{M^2}{\sum_{x,y,z} [V(x,y,z) - R(x,y,z)]^2} \right) .
\]  

Here \( V(x,y,z) \) is the reconstructed voxel grayscale value at coordinates \((x,y,z)\) scaled to the interval \([0,M]\), and \( R(x,y,z) \) is a reference voxel with the ‘correct’ result. The higher the PSNR, the more accurate the reconstruction. The RMSE is computed over the whole volume (lower is better).

There is no significant difference neither in performance nor accuracy between the regular divide and rcpps with Newton–Raphson. The performance improvement when

### Table 7. Measured performance and the deviation against the model prediction.

| Performance (GUP/s) | HPT | WEM | WEX | SNB | SNB (AVX) |
|---------------------|-----|-----|-----|-----|-----------|
| Deviation           | -13.3% | -18.3% | -11.5% | -12.6% | -18.0% |

### Table 8. Root mean square error (RMSE), the peak signal-to-noise ratio (PSNR), and the performance of three code alternatives (regular divide instruction, reciprocal, and reciprocal with Newton–Raphson iteration), on the Westmere test machine for the fully optimized case (full node).

| Performance (GUP/s) | RMSE | PSNR (dB) |
|---------------------|------|-----------|
| divps               | 3.79 | 0.03      |
| rcpps               | 4.21 | 1.68      |
| Newton–Raphson      | 3.79 | 0.03      |
|                    |      | 103       |
using rcpps alone is 10%, at an accuracy which is still better than for the published GPGPU implementations (63–65 dB). In the following section we discuss performance results using this latter version. For a visual inspection of the accuracy see also Figure 7 which shows reconstructed axial slice images.

7.3 Parallel results

Figure 8(a)–(d) displays a summary of all performance results on node and socket levels, and parallel scaling inside one socket for the best version on each architecture. All machines show nearly ideal scaling inside one socket when using physical cores only. With SMT, the benefit is considerable on Sandy Bridge (33%) and Westmere (31%), and a little smaller on Westmere EX (25%). The large effect on Sandy Bridge may be attributed to a higher number of bubbles in the pipeline, as indicated by the larger discrepancy between the ‘throughput’ and ‘critical path’ cycles in the AVX loop kernel (see Section 5.2). Scalability from one to all sockets of the node is also close to perfect for the multisocket machines, with the exception of Westmere EX, on which there is a slight load imbalance due to 80 threads working on only 512 slices.

Depending on the architecture, SSE vectorization boosts performance by a factor of 2–3 on the socket level. As explained earlier (see Section 4), part 2 of the algorithm prohibits the optimal speedup of 4 because its runtime is linear in the SIMD vector length. Work reduction through clipping alone shows only limited effect due to load imbalance, but this can be remedied by an appropriate cyclic OpenMP scheduling, as described in Section 6. This kind of load balancing not only improves the work distribution but also leads to a similar access pattern to the projection images across all threads. This can be seen in Figure 9(a), which shows the cacheline traffic between the L2 and L1

---

**Figure 7.** Axial slices through the volume reconstructed with different implementations. (a)–(c) were reconstructed using the divide operation (most accurate), the reciprocal with Newton–Raphson, and only the reciprocal, respectively (value range [0, 3,039]). The difference between (a) and (b) was generally below any visual relevance. (d) shows the absolute difference between (a) and (c) (value range [0, 19.5]).
cache during a six-thread run on one Westmere socket with all optimizations except clipping (only three cores are shown for clarity). Although the amount of work, i.e. the number of voxels, is perfectly balanced across all threads, there is a strong discrepancy in cacheline traffic between threads when standard static scheduling is used. The reason for this is that the projections of voxel lines onto the detector are quite different for lines that are far apart from each other in the volume, which leads to vastly different access patterns to the image data, and hence very dissimilar locality properties. Cyclic scheduling removes this variation (see Figure 9(b)).

Cache blocking has little to no effect on all architectures except Harpertown, as predicted by our analysis. Figure 9 shows timelines for socket floating point performance and bandwidth on one Westmere socket, comparing blocked/non-blocked and SMT/non-SMT variants. Figure 10(a) clearly demonstrates the performance boost of SMT in contrast to the very limited effect of blocking. On the other hand, the blocked implementation significantly reduces the

Figure 8. Scalability and performance results for the $512^3$ test case on all platforms. In-socket scalability was tested using the best version of the SIMD-vectorized line update kernel on each system (AVX-V1 on Sandy Bridge, SSE-V2 on all others). The practical performance goal for complete reconstruction (20 seconds runtime, corresponding to 3.33 GUP/s) is indicated as a dashed line. Gflop/s numbers have been computed assuming 31 flop/s per optimized (scalar) inner loop iteration. Note the scale change between the left and right pairs of graphs.

Figure 9. Timeline view of the L2 to L1 cacheline traffic for three cores of a six-thread run without clipping on one Westmere socket, using (a) standard static OpenMP scheduling and (b) cyclic ‘static,1’ scheduling.

Figure 10. Performance counter timeline monitoring of floating-point performance (a) and memory bandwidth (b), comparing blocked/non-blocked and SMT/non-SMT variants of the best implementation on one Westmere socket at 100 ms resolution. The inset in (b) shows a zoomed-in view with 2 ms resolution.
bandwidth requirements (Figure 9(b)). The blocked variants have a noticeable amplitude of variations while the non-blocked versions appear smooth. In the inset of Figure 9(b) we show a zoomed-in view with finer time resolution, which indicates that the frequency of bandwidth variations is much larger without blocking; this is plausible since the bandwidth is dominated by the voxel volume in this case. With blocking, individual voxel lines are transferred in “bursts” with phases of inactivity in between, where image data is read at low bandwidth.

The benefit of AVX on Sandy Bridge falls short of expectations for the same reason as in the SSE case. Still it is remarkable that the desktop Sandy Bridge system outperforms the dual-socket Harpertown server node, which features twice the number of cores at a similar clock speed. Both Westmere and Westmere EX meet the performance requirements of at most 20 seconds for a complete volume reconstruction. The Westmere EX node is, however, not competitive due to its unfavorable price-to-performance ratio. It is an option if absolute performance is the only criterion.

8 CPU versus GPGPU

Since the backprojection algorithm is well suited for GPGPUs, the performance leaders of the open competition benchmark have traditionally been GPGPU implementations in OpenCL and CUDA. The gap to the fastest CPU version reported on the RabbitCT Web site (http://www.rabbitct.com/) before this work was started was very large. For the reasons given in the derivation of the performance model, simple bandwidth or peak performance comparisons are inadequate to estimate the expected reconstruction speed advantage of GPGPUs, although it should certainly lie inside the usual corridor of 4–10 when comparing with a full CPU socket. An example of a well-optimized GPU code was published recently (Papenhausen et al. 2011). Our implementation shows that current x86 multicore chips are truly competitive (see Figure 11), even when considering the price/performance ratio. Beyond the clinically relevant 512³ case, industrial applications need higher resolutions. This is a problem for GPGPU implementations because the local memory on the card is often too small to hold the complete voxel volume, causing extra overhead for moving partial volumes in and out of the local memory and leading to a more complex implementation. If the price for the hardware is unimportant, a Westmere EX node is an option that can easily outperform GPGPUs. Note that the unusually large gap between the performance levels at 512³ and 1,024³ on this architecture may be attributed to better load balancing when 80 threads work on 1024 instead of 512 slices. It also lifts the performance efficiency (fraction of node peak) on this system to about 50%, matching the level achieved by the other multicore nodes already on the smaller problem. This compares with 26% for the CUDA code on the GTX480 card, assuming the same number of flops per voxel update.

The results for the Sandy Bridge desktop system and the good scalability even of the optimized algorithm promise even better performance levels on commodity multicore systems in the future. Note that it would be possible to provide a simple and efficient distributed memory parallelization of the algorithm for even faster reconstruction. ‘Micro-clusters’ based on cheap desktop technology could then easily meet any time constraint at extremely low cost. However, a comparison based on the hardware cost alone is certainly too simplistic, especially when taking into account the overall cost for a complete CT scanner including maintenance.

9 Conclusions and outlook

We have demonstrated several algorithmic and low-level optimizations for a CT backprojection algorithm on current Intel x86 multicore processors. Highly optimizing compilers were not able to deliver useful SIMD-vectorized code. Our implementation is thus based on assembly language and vectorized using the standard instruction set extensions SSE and AVX. The results show that commodity hardware can be competitive with highly tuned GPU implementations for the clinically relevant 512³ voxel case at the same level of accuracy. Non-pipelined divide instructions (divps) or a fast pipelined version (rcpps) with subsequent Newton–Raphson iteration proved to be equivalent in terms of accuracy and performance. Compared with a pure reciprocal they provide better accuracy at a 10% performance penalty. The standard dual-socket server system Intel Westmere EP is easily able to beat the 20 second limit for the full backprojection, reaching 15.8 seconds. Preliminary tests
on an Intel Sandy Bridge EP Xeon platform (8 cores per socket) showed that runtimes close to the GPU results are in reach for modestly priced dual-socket servers.

We have demonstrated that it is necessary to consider all aspects of processor and system architecture in order to reach best performance, and that the effects of different optimizations are closely connected to each other. The benefit of the AVX instruction set on Sandy Bridge was limited due to the lack of a gathered load and the small number of instructions that natively operate on the full SIMD register width. This relevant algorithm can achieve very good efficiency on commodity processors and it would be a natural step to further improve performance with a distributed memory implementation. At higher resolutions, which are used in industrial applications, multicore systems are frequently the only choice (apart from expensive custom solutions).

Future work includes a more thorough analysis and optimization of the AVX line update kernel, and an inclusion of the new AVX2 gather operations once they become available.

Notes
1. See http://www.rabbitct.com/
2. See http://www.intel.com/technology/architecture-silicon/mic/index.htm
3. See http://www.theregister.co.uk/2011/02/25/ict_godson_3bchip/page2.html
4. See http://www.streambench.org/
5. See http://code.google.com/p/likwid
6. We use SI prexes, i.e. 1 GUP/s means 10
9 updates per second. This is inconsistent with a large part of the literature on medical image reconstruction, where ‘G’ is used as a binary prefix for 2
30  1.074.10
9 (Goddard et al. 2007).

Funding
This work was partially supported by the Bavarian Competence Network for Scientific High Performance Computing (KONWIHR) under project OM4papps.

References
Feldkamp L, Davis L and Kress J (1984) Practical cone-beam algorithm. Journal of the Optical Society of America A1(6): 612–619.

Goddard I, Berman A, Bockenbach O, Lauginiger F, Schuberth S and Thieret S (2007) Evolution of computer technology for fast cone beam backprojection. In Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series, vol. 6498.

Gregor J, Gleason SS, Paulus MJ and Cates J (2002) Fast Feldkamp reconstruction based on focus of attention and distributed computing. International Journal of Imaging Systems and Technology 12: 229–234.

Hager G and Wellein G (2010) Introduction to High Performance Computing for Scientists and Engineers. Boca Raton, FL: CRC Press.

Hartley R and Zissermann A (2004) Multiple View Geometry in Computer Vision, 2nd edn. Cambridge: Cambridge University Press.

Heigl B and Kowarschik M (2007) High-speed reconstruction for C-arm computed tomography. In 9th International Meeting on Fully Three-Dimensional Image Reconstruction in Radiology and Nuclear Medicine, Lindau, pp. 25–28, http://www.fully3d.org.

Hofmann HG, Keck B and Hornegger J (2010) Accelerated C-arm reconstruction by out-of-projection prediction. In Deserno TM, Handels H, Meinzner H-P and Tolxdorf T (eds), Bildverarbeitung für die Medizin 2010, Berlin, pp. 380–384. Available at: http://www5.informatik.uni-erlangen.de/Forschung/Publikationen/2010/Hofmann10-ACR.pdf.

Hofmann HG, Keck B, Rohkohl C and Hornegger J (2011) Comparing performance of many-core CPUs and GPUs for static and motion compensated reconstruction of C-arm CT data. Medical Physics 38: 3940–3944.

Intel (1999) AP-803 increasing the accuracy of the results from the reciprocal and reciprocal square root instructions using the Newton–Raphson method.

Intel (2011 a) Intel advanced vector extensions programming reference.

Intel (2011 b) Intel architecture code analyzer. Available at: http://software.intel.com/en-us/articles/intel-architecture-code-analyzer/.

Kachelrieß M, Knaup M and Bockenbach O (2007) Hyperfast parallel-beam and cone-beam backprojection using the CELL general purpose hardware. Medical Physics 34: 1474–1486.

Kak A and Slaney M (2001) Principles of Computerized Tomographic Imaging. Philadelphia, PA: SIAM.

Keck B, Hofmann H, Schell H, Kowarschik M and Hornegger J (2009) High resolution iterative CT reconstruction using graphics hardware. In Yu B (ed.), 2009 IEEE Nuclear Science Symposium Conference Record, pp. 4035–4040. Available at: http://www5.informatik.uni-erlangen.de/Forschung/Publikationen/2009/Keck09-HRI.pdf.

Kunze H, Häwer W and Stierstorfer K (2007) Iterative extended field of view reconstruction. In Hsieh J and Flynn MJ (eds), Medical Imaging 2007: Physics of Medical Imaging, San Diego, CA (Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series, vol. 6510).

Maier A, Choi J-H, Keil A, Niebler C, Sarmiento M, Fieselmann A, et al. (2011) Analysis of vertical and horizontal circular C-arm trajectories. In Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series, vol. 7961.

Mueller K, Xu F and Neophytou N (2007). Why do commodity graphics hardware boards (GPUs) work so well for acceleration of computed tomography? In SPIE Electronic Imaging Conference, vol. 6498, 64980N.1–64980N.12.

Mueller K and Yagel R (1998) Rapid 3D cone-beam reconstruction with the Algebraic Reconstruction Technique (ART) by utilizing texture mapping graphics hardware. Nuclear Science Symposium, 1998. Conference Record, vol. 3, pp. 1552–1559.

Navab N, Bani-Hashemi A, Nadar M, Wiesent K, Durlak P, Brunner T, et al. (1998) 3D reconstruction from projection matrices in a C-arm based 3D-angiography system. In Wells W, Colchester A and Delp S (eds), Medical Image Computing and Computer-Assisted Intervention MICCAI 98 (Lecture Notes in Computer Science, vol. 1496). Berlin: Springer, pp. 119–129.
Okitu Y, Ino F, and Hagihara K (2010). High-performance cone beam reconstruction using CUDA compatible GPUs. Parallel Computing 36: 129–141.

Papenhausen E, Zheng Z and Mueller K (2011) GPU-accelerated back-projection revisited: squeezing performance by careful tuning. In Workshop on High Performance Image Reconstruction (HPIR).

Rohkohl C, Keck B, Hofmann HG and Hornegger J (2009 a) Rabbit CT—an open platform for benchmarking 3-D cone-beam reconstruction algorithms. Medical Physics 36: 3940–3944.

Rohkohl C, Lauritsch G, Prümmer M and Hornegger J (2009 b) Interventional 4-D motion estimation and reconstruction of cardiac vasculature without motion periodicity assumption. In Yang G-Z, Hawkes D, Rueckert D, Noble A and Taylor C (eds), Medical Image Computing and Computer-Assisted Intervention – MICCAI 2009 (Lecture Notes in Computer Science, vol. 5761). Berlin: Springer, pp. 132–139.

Scherl H, Keck B, Kowarschik M and Hornegger J (2009 a) Fast GPU-Based CT Reconstruction using the Common Unified Device Architecture (CUDA). In Frey EC (ed.), IEEE Nuclear Science Symposium Conference Record, 2007 (NSS ’07), Honolulu, HI, vol. 6, pp. 4464–4466.

Scherl H, Koerner M, Hofmann H, Eckert W, Kowarschik M and Hornegger J (2007 b) Implementation of the FDK algorithm for cone-beam CT on the cell broadband engine architecture. In Hsieh J and Flynn MJ (eds), SPIE Medical Imaging Conference Proceedings, vol. 6510. SPIE.

Strobel N, et al. (2009) 3D Imaging with Flat-Detector C-Arm Systems. In Multislice CT, 3rd edn. Berlin: Springer, pp. 33–51.

Treibig J and Hager G (2009) Introducing a performance model for bandwidth-limited loop kernels. In Proceedings of the Workshop Memory issues on Multi- and Manycore Platforms at PPAM 2009, the 8th International Conference on Parallel Processing and Applied Mathematics, Wroclaw, Poland.

Treibig J, Hager G and Wellein G (2009) Complexities of performance prediction for bandwidth-limited loop kernels on multicore architectures. In High Performance Computing in Science and Engineering. Berlin: Springer.

Treibig J, Hager G and Wellein G (2010) Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In Proceedings of PSTI2010, the First International Workshop on Parallel Software Tools and Tool Infrastructures, San Diego, CA.

Wiesent K, Barth K, Navab N, Durlak P, Brunner T, Schuetz O, et al. (2000) Enhanced 3-D-reconstruction algorithm for C-arm systems suitable for interventional procedures. IEEE Transactions on Medical Imaging 19: 391–403.

Zeng K, Bai E and Wang G (2007) A fast CT reconstruction scheme for a general multicore PC. International Journal of Biomedical Imaging 2007: 29160.

Author biographies

Jan Treibig holds a PhD in Computer Science from the University of Erlangen-Nuremberg. From 2006 to 2008 he was a software developer and quality engineer in the embedded automotive software industry. Since 2008 he has been a research scientist in the HPC Services group at Erlangen Regional Computing Center (RRZE). His main research interests are low-level and architecture-specific optimization, performance modeling, and tooling for performance-oriented software developers.

Georg Hager holds a PhD in Computational Physics from the University of Greifswald. He has been working with high performance systems since 1995, and is now a senior research scientist in the HPC group at Erlangen Regional Computing Center (RRZE). Recent research includes architecture-specific optimization for current microprocessors, performance modeling on processor and system levels, and the efficient use of hybrid parallel systems. His daily work encompasses all aspects of user support in HPC such as lectures, tutorials, training, code parallelization, profiling and optimization, and the assessment of novel computer architectures and tools.

Hannes G. Hofmann is grad student at the Pattern Recognition Lab at the Friedrich-Alexander-University Erlangen-Nuremberg, Germany. He received his BSc and MSc degrees in computer science from the University of Erlangen-Nuremberg in 2004 and 2008, respectively. His research interest is hardware acceleration of medical image processing algorithms.

Joachim Hornegger is professor and director of the Pattern Recognition Lab at the Friedrich-Alexander-University Erlangen-Nuremberg, Germany. He received his MSc in theoretical computer science/mathematics in 1992 and his PhD in applied computer science in 1996. His PhD thesis was on statistical learning, recognition, and pose estimation of 3D objects. He was an assistant professor at the University of Erlangen-Nuremberg (1996–1997), a visiting scholar and lecturer at Stanford University (Stanford, CA, USA) in the academic year 1997–1998, and served as a lecturer at the universities of Erlangen, Eichstaett-Ingolstadt, and Mannheim. From 1998 to 2003 he was director of medical image processing at Siemens Healthcare.

Gerhard Wellein holds a PhD in Solid State Physics from the University of Bayreuth and is a regular Professor at the Department for Computer Science at University of Erlangen. He heads the HPC group at Erlangen Regional Computing Center (RRZE) and has more than 10 years of experience in teaching HPC techniques to students and scientists from Computational Science and Engineering. His research interests include solving large sparse eigenvalue problems, novel parallelization approaches, performance modeling, and architecture-specific optimization.