J Supercomput (2011) 57:151–160
DOI 10.1007/s11227-010-0397-z

Parallel medical image reconstruction: from graphics processing units (GPU) to Grids

Maraike Schellmann · Sergei Gorlatch · Dominik Meiländer · Thomas Kösters · Klaus Schäfers · Frank Wübbeling · Martin Burger

Published online: 5 March 2010

Abstract We present and compare a variety of parallelization approaches for a real-world case study on modern parallel and distributed computer architectures. Our case study is a production-quality, time-intensive algorithm for medical image reconstruction used in computer tomography (PET). We parallelize this algorithm for the main kinds of contemporary parallel architectures: shared-memory multiprocessors, distributed-memory clusters, graphics processing units (GPU) using the CUDA framework, the Cell processor and, finally, how various architectures can be accessed in a distributed Grid environment. The main contribution of the paper, besides the parallelization approaches, is their systematic comparison regarding four important criteria: performance, programming comfort, accessibility, and cost-effectiveness. We report results of experiments on particular parallel machines of different architectures that confirm the findings of our systematic comparison.

Keywords Medical image reconstruction · Parallel programming · Parallel architecture comparison · Positron Emission Tomography (PET) · List-mode OSEM algorithm · Cell processor · Graphics processing units (GPU) · CUDA

1 Introduction

The research presented in this paper was conducted at the interdisciplinary collaborative research center (SFB) "Molecular Cardiovascular Imaging" at the University of Münster, Germany.

M. Schellmann · S. Gorlatch · D. Meiländer (✉) · T. Kösters · K. Schäfers · F. Wübbeling · M. Burger
Institut für Informatik, Universität Münster, Einsteinstr. 62, 48149 Münster, Germany
e-mail: d.meil@uni-muenster.de

S. Gorlatch
e-mail: gorlatch@uni-muenster.de

We aim at improving the resolution and quality of Positron Emission Tomography (PET) images. Today, the runtime on an off-the-shelf computer of one of the most accurate 3D PET reconstruction algorithms (the list-mode OSEM) ranges from one hour to several days. Therefore, parallelization is crucial in order for such hardware and software techniques to be used in clinical routine.With more advanced equipment and more precise imaging algorithms, an efficient parallel implementation will become even more important.

In this paper, we focus on the parallelization of the list-mode OSEM (Ordered Subset Expectation Maximization) algorithm [7] for PET image reconstruction which is representative for a large class of modern image reconstruction methods. The foremost goal of this work is to find the most suitable parallel architecture for this algorithm. The suitability of a particular architecture is usually defined using two criteria: (1) the algorithm's parallel performance, and (2) the usability of available programming environments for this particular architecture. In medical imaging, the medical personnel will be reluctant to use parallel software if it requires a high administrative effort like searching for a free time-slot on a number of cluster computers. Therefore, our third comparison criterion will be accessibility. Moreover, while a server with several multi-core processors might provide high performance and can be easily accessed over a local file system, the purchase cost of such a server might limit its usage. Thus, we introduce cost-effectiveness as our fourth criterion.

The contribution of this paper is threefold: (1) It gives an overview of parallel implementations of the list-mode OSEM algorithm for the 3D PET image reconstruction on practically all currently available modern parallel architectures, including: shared-memory multiprocessors, multi-core processors, cluster computers, graphics hardware (GPU) and the Cell processor. (2) It identifies the most suitable parallel architecture for typical image reconstruction tasks by analyzing the parallel implementations for performance, appropriateness of programming environments, accessibility of the corresponding architecture and cost-effectiveness. (3) It outlines our distributed grid-like system which chooses the most suitable of the available parallel machines for a given imaging task and thus frees the medical personnel from all administrative efforts.

2 Iterative PET image reconstruction

In Positron Emission Tomography (PET), a radioactive substance is injected into a human or animal body. Afterwards, the body is placed inside a PET scanner that contains several arrays of detectors. As the particles of the applied substance decay, positrons are emitted (hence the name PET) and annihilate with nearby electrons. During one such annihilation, two photons are emitted in opposite directions. The "decay events" are registered by two opposite detectors at the same time. The scanner records these events in a list with each record comprising the positions of those two detectors.

For our comparative study, we consider the following representative algorithm for creating an image from the events. List-Mode Ordered Subset Expectation Maximization [7, 14] (called list-mode OSEM in the sequel) is a block-iterative algorithm

```
for (int l = 0; l < subsets; l++) {
  /* read subset */

  /* compute c_l */
  for (int i = 0; i < subset_size; i++) {
    ... }

  /* compute f_l+1 */
  for (int k = 0 ; k < image_size; k++) {
    if (c_l[k] > 0.0)
      f[k] *= c_l[k];
  } }
```

Listing 1 Sequential code comprises an outer loop with two nested inner loops

for 3D image reconstruction. List-mode OSEM takes a set of events and splits them into s equally sized subsets.

For each subset $l \in 0, \ldots, s-1$, the following computation is performed:

$$f_{l+1} = f_l c_l; \quad c_l = \frac{1}{A_N^t \mathbf{1}} \sum_{i \in S_l} (A_i)^t \frac{1}{A_i f_l}. \tag{1}$$

Here $f \in \mathbb{R}^n$ is a 3D image in vector form with dimensions $n = (X \times Y \times Z)$, $A \in \mathbb{R}^{m \times n}$, element a_{ik} of row A_i is the length of intersection of the line between the two detectors of event i with voxel k of the reconstruction region, computed using Siddon's algorithm [15]. $\frac{1}{A_N^t \mathbf{1}}$ is the so-called normalization vector. Since it can be precomputed, we will omit it in the following. Note that the multiplication of $f_l c_l$ is performed element-by-element. Each subset's computation takes its predecessor's output image as input and produces a new, more precise image.

The overall structure of the sequential list-mode OSEM implementation comprises three nested loops: one outer loop with two inner loops. The outer loop iterates over the subsets. The first inner loop iterates over a subset's events to compute the summation part of c_l. The second inner loop iterates over all elements of f_l and c_l to compute f_{l+1} (Listing 1).

The algorithm studied here can be used to reconstruct data from virtually every PET scanner if a conversion method from the scanner data to world coordinates is available. In our experiments, we use data acquired by the *quadHIDAC* scanner and employ the conversion method for this scanner introduced in [7].

List-mode OSEM is a rather time-consuming algorithm. A typical 3D image reconstruction processing 6×10^7 input events for a $150 \times 150 \times 280$ PET image takes more than two hours on an off-the-shelf PC. To reduce the algorithm's runtime we developed several parallel implementations [4, 11] which we systematically compare with respect to the four criteria formulated in the introduction.

3 Parallel image reconstruction

Because of the data dependency between the subsets' computations in (1), implied by $f_{l+1} = f_l c_l$, the subsets cannot be processed in parallel. The computation of c_l and f_{l+1} is parallelizable, using the following idea.

For the computation of c_l, all parallel implementations distribute the events among the processing units (either processors or cores, from now on called PUs). Now each PU computes a partial sum of c_l. Afterwards, all partial results are summed up over the communication link. For the computation of $f_{l+1} = f_l c_l$, the image is distributed among the PUs, thus each PU computes $f_{l+1} = f_l c_l$ for its sub-image in parallel.

Parallelization on shared-memory processors On shared-memory multiprocessors and multi-core processors, we developed an OpenMP implementation following the parallelization idea described above.

To parallelize the computation of c_l and f_{l+1}, we have to parallelize the two inner loops of the list-mode OSEM algorithm. We use the `parallel for` directive of OpenMP that declares the succession for loop to be executed in parallel by a team of threads for both loops. Apart from the additional compiler directives, no considerable changes were made to the sequential program. Thus, an OpenMP-based parallel implementation of the list-mode OSEM algorithm is easily derived from a sequential implementation.

Within the first inner loop (summation part of c_l), all threads perform multiple additions to arbitrary voxels of a common intermediate image. We prevent race conditions using a mutex that declares the summation part mutually exclusive, such that only one thread at a time is able to work on the image. In OpenMP, mutexes are declared by using the *critical* construct which specifies a mutual exclusion for the successive code section.

Parallelization on cluster computers On distributed-memory clusters, we use MPI (Message Passing Interface) for the parallel implementation. Here, every process first reads "its" events from the remote file system. All processes compute their partial sum of c_l simultaneously; then the result is summed up using `MPI_Allreduce`. Finally, before the next subset is started, all processes compute f_{l+1}. Note that for the computation of f_{l+1} the image is not distributed among the processes, because the resulting network communication is more time-consuming than the actual computations.

On hybrid machines (clusters), where each node is either a shared-memory multiprocessor or a multi-core processor, we combine the MPI distributed-memory implementation with the OpenMP shared-memory implementation. Thus the partial sums of c_l and f_{l+1} are computed simultaneously by all PUs of the shared-memory machines.

Parallelization on graphics processing units (GPU) Modern GPUs (Graphics Processing Units) can be used as mathematical coprocessors: they add computing power to the CPU. A GPU is a parallel machine that consists of SIMD (Single Instruction Multiple Data) multiprocessors (ranging from 1 to 32). The stream processors of a SIMD multiprocessor are called shader units. The GPU (also called *device*)

has its own fast memory with an amount of up to 4 GB. On the main board, one to four GPUs can be installed and used as coprocessors simultaneously.

With CUDA (Compute Unified Device Architecture) [2], the GPU vendor NVIDIA provides a programming interface that introduces the thread-programming concept for GPUs to the C programming language. A block of threads executing the same code fragment, the so-called *kernel* program, runs on one multiprocessor. Each thread of this block runs on one of the shader units of the GPU, each unit executing the kernel on a different data element. All blocks of threads of one application are distributed among the multiprocessors by the *scheduler*. The GPU's device memory is shared among all threads.

The calculations for one subset in our GPU implementation proceed as follows:

1. The CPU reads the subsets' events and copies them to the GPU device memory.
2. Each thread computes a partial sum of c_l and adds it directly to the device memory. The amount of events per thread is chosen according to the following considerations: Firstly, as many threads as possible should be started in order to hide memory latency efficiently [8]. However, each thread needs to save partial results in the device memory, which requires too much memory if one thread is started per event. Therefore, the maximum number of threads is started so that all partial results still fit into the device memory.
3. Each thread computes one voxel value for $f_{l+1} = f_l c_l$.
4. f_{l+1} is copied back to the CPU.

Note that during the computation of c_l (step 2), the threads write, as in the shared-memory implementation, directly to the shared vector c_l. In order to avoid race conditions, we again have to protect c_l with a mutex. Since this is not directly possible with CUDA (necessary mechanisms are lacking, only atomic `integer` operations exist), we decided to allow race conditions in the GPU implementation in cases where quantitative results are not required (see [13] for details). For quantitative experiments, we can use a thread-safe shared-memory or Cell processor reconstruction.

When we use two GPUs at the same time, we have two separate device memories. The computations proceed as above, with each GPU computing half of the events during the forward-projections (step 2) and half of the sub-images during the computation of f_{l+1} (step 3). After all forward-projections, the two c_ls residing on the device memories need to be summed up.

Parallelization on the cell processor The Cell Broadband Engine is a multiprocessor developed jointly by Sony Computer Entertainment Inc., Toshiba Corp. and IBM Corp. It consists of one PowerPC Processor Element (*PPE*) and eight processing cores called Synergistic Processor Elements (*SPEs*). Communication is performed through the Element Interconnection Bus (*EIB*). To program applications for the Cell processor, IBM provides a Software Development Kit (SDK) [1] which contains the GCC C/C++-language compilers for the PPU and the SPU.

The calculation of one subiteration of our example algorithm on p SPEs proceeds as follows:

1. The PPE reads the subsets' events and stores them in the main storage. Afterwards, the PPE sends each SPE a message to start computations.

2. Each thread computes a partial sum of c_l and adds it directly to the device memory. Since all threads write simultaneously to the shared c_l, we use an atomic operation.
3. The reconstruction image is divided into sub-images f^j. Each SPE computes $f_{l+1}^j = f_l^j \, c_l^j$ on its sub-image.

Note that for the forward projection (step 2), the programmer has to organize transferring the required voxels of f_l and c_l from main storage to the SPEs' local store. Since the minimum DMA transfer size is 128 bytes, then 128 bytes instead of 4 bytes for one `float` have to be transferred for each voxel of f_l, when computing $c_{l,j}$ and $c_l + c_{l,j}$. Since a path, in almost all cases, crosses through several y- and z-planes of the 3D image, the bulk of additional transferred voxels of f_l cannot be used in the following computations. Using the minimum transfer size of 128 bytes, an average of 1.6 of the 32 transferred voxels, i.e., 5% of each DMA transfer, are used.

When using two Cell processors, i.e., two PPEs and altogether sixteen SPEs, we have two main storages, such that each PPE only communicates with its SPEs. The communication between both main storages and the SPE management is transparent to the programmer and thus the programmer can develop his code as if there were only one PPE with sixteen SPEs.

Grid system for PET reconstruction MIRGrid (Medical Image Reconstruction Grid) [10] is an experimental grid system that we have developed to integrate in a single application all steps of the imaging process, which are traditionally performed by the user using different software tools: from reading the raw data acquired by the scanner, over transparent parallel reconstruction to the visualization and storage of reconstructed images.

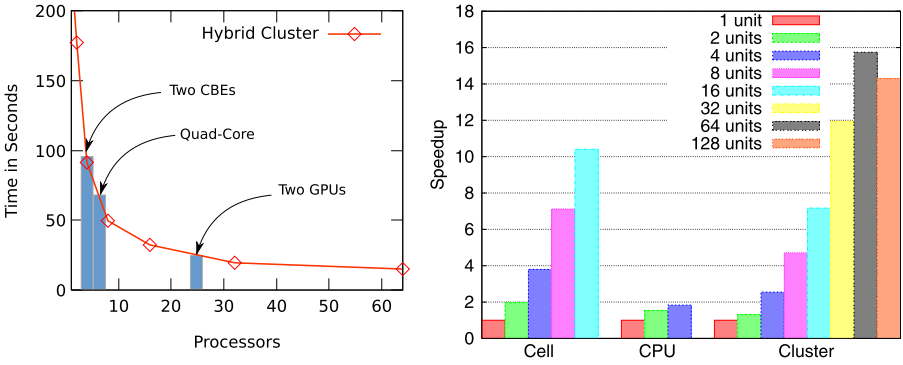
After the user has chosen the raw data previously collected by the scanner and the parameters for reconstruction, the client sends the data and the parameters to the scheduler. Transparently to the user, the MIRGrid scheduler then assigns the reconstruction to an HPC, and the runtime system starts and monitors the reconstruction on that HPC. When the reconstruction is finished, the result images are sent back to the client where they are visualized and stored.

The MIRGrid system currently supports shared-memory machines and cluster computers. The system is installed at the nuclear medicine clinic in Münster and is currently tested before going into productive use in a few months. We plan to integrate support for GPUs on MIRGrid in the near future.

4 Runtime experiments and architecture comparison

Since the image size of the list-mode OSEM on all four architectures presented in the previous section has only little influence on scalability [6, 12], we restrict our considerations to the typical image size of $N = (150 \times 150 \times 280)$. We use 10^7 events in 10 subsets acquired during a 15-minute mouse scan of the quadHIDAC [9] small-animal PET scanner.

We use the following parallel machines in our experiments:



(a) Runtime (b) Speedup

Fig. 1 *Left*: Runtime comparison of hybrid cluster (*curve*); quad-core processor, two GPUs and two Cell processors (*bars*). *Right*: Speedup comparison of hybrid cluster (units $\widehat{=}$ processors), quad-core processor (units $\widehat{=}$ cores) and two Cell processors (units $\widehat{=}$ SPEs)

Quad-core Processor: Intel Core 2 Quad processor with four cores running at 2.83 GHz. Two cores share 6 MB level 2 cache and all cores share the 4 GB main memory. The memory throughput is up to 11 GB/s.

Hybrid Cluster: 200 Dual INTEL Xeon 3.2 GHz 64 bit nodes, each with 4 GByte main memory, connected by an InfiniBand network. To exploit the fast InfiniBand interconnect (point-to-point throughput of up to 900 MB/s), we used the Scali MPI Connect implementation on this machine.

GPU: Two NVIDIA GeForce 8800 GTX which have 16 SIMD-multiprocessors, each with 8 shader units running at 1.35 GHz. The device memory is 768 MB. The measured throughput between device and CPU main memory is 1.5 GB/s. The multi-processor to device throughput is 86 GB/s.

Cell Processor: A QS21 Blade Center equipped with two Cell processors. Each Cell processor consists of one PPE running at 3.2 GHz with 512 KB L2 cache and 1 GB main memory and 8 SPEs running also at 3.2 GHz equipped with 256 KB local storage. The EIB supports a peak bandwidth of 204.8 GB/s and the integrated memory controller (MIC) provides a peak bandwidth of 25.6 GB/s to the DDR2 memory.

Performance In the following, we analyze the performance of the parallel implementation in terms of total runtime (Fig. 1a) and scalability (Fig. 1b).

The hybrid cluster outperforms all other architectures with a minimum reconstruction time of ≈15 seconds on 64 processors. However, the implementation does not scale well: the speedup on 16 processors is ≈7 and thus less than 50% of the ideal speedup. Moreover, runtime deteriorates for 128 processors. Refer to [4] for a detailed scalability analysis.

The two GPUs are only 1.6 times slower than the 64-processor cluster. However, runtime only decreased from 33 seconds to 24 seconds when going from one to two GPUs. Since this is less than 50%, we can only expect little speedup by adding more GPUs over the main board's PCI Express slots. Furthermore, since only an insufficient profiling tool exists, it is quite difficult to assess what the current performance

Table 1 Average measured runtime of the list-mode OSEM algorithm for 10^7 events in 10 subiterations and estimated purchase price for the corresponding architecture

Architecture	Multi-core 4 cores	Hybrid 64 processors	GPU 2 devices	Cell 2 cell procs
Runtime:	72.6 s	14.8 s	24.4 s	99.8 s
Est. Price:	€1.500	€1.500.000	€2.000	€5.500

bottleneck is. Therefore, we cannot determine if more shader units or increased memory bandwidth would speed up our application.

The quad-core processor is ≈5 times slower than the 64 processors of the hybrid cluster. The main limiting scalability factor on the quad-core processor is, as on the cluster, the restricted memory bandwidth.

Although four times as many cores are available in two Cell processors with overall 16 SPEs as on the quad-core processor, two Cells still provide the worst runtime. As described in Sect. 3, only 5% of each 128 byte DMA transfer is actually used in computations. Therefore, a lot of time is spent in transferring large amounts of unused data. Hence, the minimum size of 128 bytes per DMA transfer is an important limiting factor of the Cell architecture in our application.

Programming comfort Today, all four programming tools we use to implement the parallel algorithm, can be seen as the standard to program the according architectures. Therefore, comparing the programming tools allows us to compare the architectures with respect to their programmability. In [13] we explain why the Cell SDK provides the lowest abstraction level. We also show why CUDA provides a higher abstraction level, but its lack of sufficient debugging tools makes programming GPUs with this framework more tedious than programming with MPI. Finally, OpenMP provides the highest abstraction level and is the easiest to use.

Accessibility Multi-core computers, workstations with CUDA-enabled GPUs and a Cell blade can be run in a local network. Therefore, accessibility is high for all three. This is especially true for multi-core processors, because they are available in virtually every off-the-shelf computer today. On the contrary, buying a cluster for a medical clinic will most likely be too expensive (Table 1). Furthermore, accessing a remote cluster results in two problems: (1) additional administrative effort is necessary in order to reconstruct images on a remote cluster, e.g., the locating of free resources, and (2) input and output data have to be transferred over the Internet from and to the cluster. While the first problem can be solved with the grid system we introduced in Sect. 3, the second problem leads to considerably longer runtime.

Cost-effectiveness The Cell processor demonstrates rather poor performance for our algorithm and is not cheaper than GPUs and multi-core processors; thus, it is less cost-effective. We estimate a workstation equipped with a high-end CPU and a low-cost GPU to be about as expensive as a workstation with a medium-cost CPU and two high-end GPUs. But since the GPU outperforms the multi-core CPU by a factor of two, the GPU is more cost-effective. Buying and maintaining a cluster is quite expensive (about 1.5 million euros for the cluster used in our experiments). Therefore,

a cluster is definitely less cost-effective than the other options. Summarizing, GPU proves to be the most cost-effective parallel architecture, followed by the multi-core CPU.

5 Conclusion

Our comparison of different parallelization approaches for an important medical imaging application has brought several important findings. The GPU proved to be the most cost-effective architecture. Since it is also quite well accessible, it is suitable for standard image reconstruction tasks. However, if very accurate quantitative reconstruction results are required, then multi-core processors or hybrid clusters are to be preferred, because, in contrast to the GPU, they allow to prevent race conditions. Also, programming for the GPU is quite tedious and error-prone; therefore, for research code that is continuously enhanced and tested, multi-core processors with OpenMP are to be preferred.

Currently, new algorithms are being developed in our collaborative research group that are even more compute-intensive than the standard list-mode OSEM. For example, we estimate that the so-called *EM-TV* algorithm [3] applied to 3D PET and an advanced scatter correction method [5] will both be one to two orders of magnitude more compute-intensive than the current algorithm. In order to use such algorithms, clusters of multi-core processors will probably be the architecture to target.

Our analysis and experiments demonstrated that the Cell processor is not very useful for the list-mode OSEM reconstruction, because it provides poor performance and is quite difficult to program.

Finally, our MIRGrid system transparently chooses the most suitable architecture for a given reconstruction task from the set of available parallel machines.

Acknowledgements We thank the NVIDIA corporation for the donation of the graphics hardware used in our experiments. We thank IBM Deutschland for letting us access their QS21 Cell blades. This work was partly funded by the Deutsche Forschungsgemeinschaft, SFB 656 MoBil (Projects B2, B3, PM6).

References

1. IBM Inc (2010) Software Development Kit for Multicore Acceleration Version 3.0. http://www.ibm.com/developerworks/power/cell/
2. NVIDIA Corp (2010) NVIDIA CUDA compute unified device architecture. http://developer.nvidia.com/object/cuda.html
3. Brune C, Sawatzky A, Burger M (2009) Bregman-EM-TV methods with application to optical nanoscopy. In: Proceedings of the 2nd international conference on scale space and variational methods in computer vision. Lectures notes in computer science, vol 5567. Springer, Berlin, pp 235–246
4. Hoefler T, Schellmann M, Gorlatch S, Lumsdaine A (2008) Communication optimization for medical image reconstruction algorithms. In: Recent advances in parallel virtual machine and message passing interface. Lectures notes in computer science, vol 5205. Springer, Berlin, pp 75–83
5. Kösters T, Wübbeling F, Natterer F (2006) Scatter correction in PET using the transport equation. In: IEEE nuclear science symposium and medical imaging conference record. IEEE, New York, pp 3305–3309
6. Meiländer D, Schellmann M, Gorlatch S (2009) Implementing a data-parallel application with low data locality on multicore processors. In: International conference on architecture of computing systems — workshop proceedings. VDE, pp 57–64

7. Reader AJ, Erlandsson K, Flower MA, Ott RJ (1998) Fast accurate iterative reconstruction for low-statistics positron volume imaging. Phys Med Biol 43(4):823–834
8. Ryoo S, Rodrigues C, Baghsorkhi S, Stone S, Kirk D, Hwu W (2008) Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In: PPoPP '08: proc of the 13th ACM SIGPLAN symposium, pp 73–82
9. Schäfers KP, Reader AJ, Kriens M, Knoess C, Schober O, Schäfers M (2005) Performance evaluation of the 32-module QuadHIDAC small-animal PET scanner. J Nucl Med 46(6):996–1004
10. Schellmann M, Böhm D, Wichmann S, Gorlatch S (2007) Towards a grid system for medical image reconstruction. In: IEEE nuclear science symposium and medical imaging conference record. IEEE, New York, pp 3019–3025
11. Schellmann M, Gorlatch S (2007) Comparison of two decomposition strategies for parallelizing the 3D list-mode OSEM algorithm. In: Proceedings fully 3D meeting and HPIR workshop, pp 37–40
12. Schellmann M, Vörding J, Gorlatch S, Meiländer D (2008) Cost-effective medical image reconstruction: from clusters to graphics processing units. In: Proceedings of the 2008 conference on computing frontiers. ACM, New York, pp 283–292
13. Schelmann M, Gorlatch M, Meiländer D, Kösters T, Schäfers K, Wübbeling F, Burger M (2009) Parallel medical image reconstruction: from graphics processors to grids. In: 10th International Conference PaCT-2009. Lectures notes in computer science, vol 5698. Springer, Berlin, pp 457–473
14. Shepp LA, Vardi Y (1982) Maximum likelihood reconstruction for emission tomography. IEEE Trans Med Imag 1:113–122
15. Siddon RL (1985) Fast calculation of the exact radiological path for a three-dimensional CT array. Med Phys 12(2):252–255