Systematic Parallelization of Medical Image Reconstruction for Graphics Hardware

Maraike Schellmann, Jürgen Vörding, and Sergei Gorlatch

University of Münster, Germany
{schellmann,voerding,gorlatch}@uni-muenster.de

Abstract. Modern Graphics Processing Units (GPUs) consist of several SIMD-processors and thus provide a high degree of parallelism at low cost. We introduce a new approach to systematically develop parallel image reconstruction algorithms for GPUs from their parallel equivalents for distributed-memory machines. We use High-Level Petri Nets (HLPN) to intuitively describe the parallel implementations for distributed-memory machines. By denoting the functions of the HLPN with memory requirements and information about data distribution, we are able to identify parallel functions that can be implemented efficiently on the GPU. For an important iterative medical image reconstruction algorithm —the list-mode OSEM algorithm—we demonstrate the limitations of its distributed-memory implementation and show how our HLPN-based approach leads to a fast implementation on GPUs, reusable across different medical imaging devices.

1 Introduction

In order to achieve good scalability, time-consuming numerical algorithms need to be parallelized on architectures with both a high degree of parallelism and large memory bandwidth. One such architecture is the modern Graphics Processing Unit (GPU) that comprises one or several SIMD (Single Instruction Multiple Data) processors. High-level languages for the general purpose GPU programming like CUDA [3] (Compute Unified Device Architecture) and BrookGPU [1] have emerged recently; they allow developers to write reasonably fast code for the GPU without dealing with the details of the underlying hardware. However, implementing GPU algorithms from scratch is often error-prone and results in implementations that lack modularity and reusability [5].

We focus on the GPU parallelization problem for iterative medical image reconstruction algorithms that solve large, sparse linear systems for a 3D reconstruction image, with a potential to extend our approach to a broader class of numerical algorithms, including iterative linear system solvers. In our approach, we use already existing parallel algorithms for distributed-memory architectures to identify parts that can be re-implemented efficiently in a data-parallel manner on the GPU. We use High-Level Petri Nets (HLPN) to describe parallel algorithms in a simple, intuitive way. We then annotate the HLPN with memory requirements and the type of data distribution that is used in the distributed-memory

E. Luque, T. Margalef, and D. Benítez (Eds.): Euro-Par 2008, LNCS 5168, pp. 811–821, 2008.

algorithm. This additional information allows us to identify data-parallelism (which is needed in order to use the SIMD processors efficiently) and memory requirements (which are important due to the comparatively small amount of memory available on the GPU) for the parallel functions.

All iterative medical image reconstruction algorithms share similar computation and communication patterns, because they are usually either multiplicative or additive versions of the Kaczmarz method [4]. Hence, our approach can be applied to all such algorithms. As an example algorithm which we parallelize for GPUs using our Petri Net approach, we use a very accurate, but also quite time-consuming algorithm used in PET (Positron Emission Tomography) reconstruction: the list-mode OSEM (Ordered Subset Expectation Maximization) [7]. We present two parallel strategies for this algorithm on the distributed-memory architecture and show their limitations. By using the Petri Net approach to reimplement these parallel strategies on the GPU, we overcome the limitations of the distributed-memory architecture and, at the same time, facilitate code reusability and modularity.

In medical image reconstruction, the lack of reusability of GPU parallelizations becomes very critical: Previous GPU implementations of the list-mode OSEM algorithm (as shown in [6]), as well as implementations for other iterative reconstruction techniques in medical image reconstruction (see [12]), have focused on the parallelization within the so-called *projection* step. This leads to poor reusability of the code: Different PET devices require different projections, and therefore, for each new device type, a new projection and thus a new GPU parallelization has to be implemented.

The remainder of this paper is structured as follows: we start with an introduction to PET and the list-mode OSEM algorithm in Section 2. We then describe the parallelization of the list-mode OSEM algorithm on distributed-memory computers in Section 3. In Section 4, we introduce our main contribution, the new Petri Net approach, and describe its application to the strategies developed in Section 3. We present experimental results in Section 5 and finally conclude the paper in Section 6.

2 PET and the List-Mode OSEM Algorithm

Positron Emission Tomography (*PET*) is a medical imaging technique that displays metabolic processes in a human or animal body. The data for reconstruction are collected in the PET acquisition process as follows. A slightly radioactive substance which emits positrons is applied to the patient who is placed inside a *scanner*. Each scanner type has a different number of detectors that can be arranged either on rings or banks surrounding the patient. The detectors of a scanner measure so-called *events*: When the emitted positrons of the radioactive substance collide with an electron residing in the surrounding tissue near the decaying spot, they are annihilated. During annihilation, two gamma rays emit from the annihilation spot in opposite directions and form a line, see Fig. 1. For each gamma ray pair, one event, i.e., the two involved detector elements, is saved.

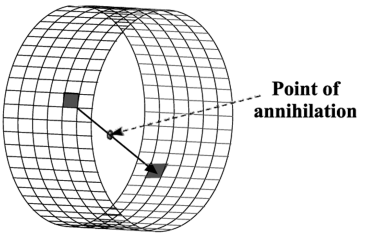


Fig. 1. Detectors register an event in a PET-scanner with 6 detector rings

During one investigation, typically $5 \cdot 10^7$ to $5 \cdot 10^8$ events are registered. From these events, a reconstruction algorithm computes a 3D image of the substance distribution in the body.

We focus in this work on the list-mode OSEM reconstruction algorithm [7] which is an enhanced version of the list-mode Expectation Maximization (EM) algorithm. EM solves the overdetermined linear system $A\,f = \mathbf{1}$ iteratively for the reconstruction image f, where $\mathbf{1} = (1, ..., 1)$ and A is a matrix with elements $a_{i,j}$ estimating the probability that the emission corresponding to event i has happened in voxel j. The computation of one row A_i of matrix A for event i is called *projection*. In general, for each estimate $a_{i,j} \in A_i$, the amount of intersection of voxel j with the line between the centers of the two detectors of event i is computed on the fly during each iteration. However, more accurate projection algorithms take into account, among others, the detector shape and gamma ray scatter inside the detector. Therefore, for each scanner, there is a different most accurate projection algorithm and it is thus critical for an implementation to be able to easily interchange projection algorithms; such implementations are called *reusable*.

The list-mode OSEM algorithm speeds up the notoriously slow EM algorithm by computing several image updates per iteration: the input dataset—consisting of all measured events—is divided into s equally-sized blocks of events, so-called subsets. The starting image vector is $f_0 = (1, ..., 1) \in \mathbb{R}^N$ (N is the number of voxels in the reconstruction image). For each subset $l \in 0, ..., s-1$, a new, more precise reconstruction image f_{l+1} is computed and used iteratively for the next subset computation as follows:

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

where S_l are the indices of events in subset l. The normalization vector $\frac{1}{A^t_{norm}\mathbf{1}}$ is independent of the current subset and can thus be precalculated. After one iteration over all subsets, the reconstruction process can either be stopped, or the result can be improved with further iterations.

3 Distributed-Memory Parallelization

We start with the two strategies for the distributed-memory parallelization of the list-mode OSEM algorithm which we analyzed in [11]:

1. the PSD (Projection Space Decomposition) strategy and
2. the ISD (Image Space Decomposition) strategy.

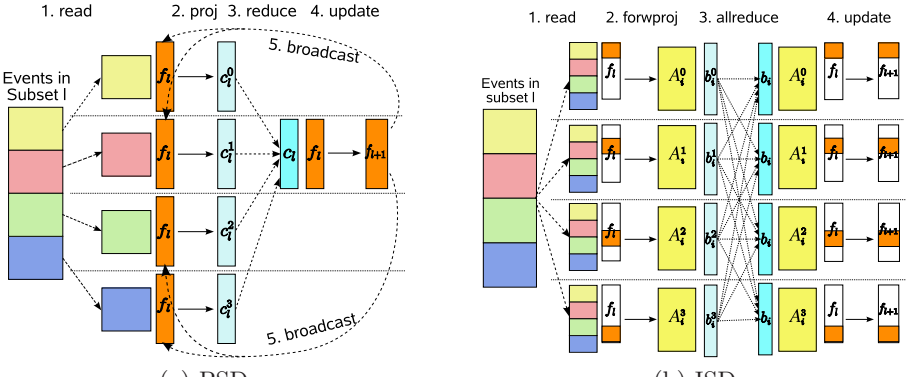
Since f_{l+1} depends on f_l in (1), both strategies need to parallelize the computations within one subset.

In the PSD (Projection Space Decomposition) strategy, we decompose the input data, i.e., the events of one subset, into p (=number of processors) blocks and compute the forward projection for these blocks simultaneously. The calculations for one subset proceed in five steps, see Fig. 2(a):

- **read:** Every processor reads its part of the subsets' events.
- **proj:** All processors k_j compute simultaneously $c_{l,j} = \sum_{i \in S_{l,j}} (A_i)^t \frac{1}{A_i f_l}$.
- **reduce:** The processors' results $c_{l,j}$ are summed up over the network.
- **update:** $f_{l+1} = f_l c_l$ is computed on one processor.
- **broadcast:** f_{l+1} is sent to all other processors.

In the Image Space Decomposition (ISD) strategy, the output data, i.e., the reconstruction image, is decomposed into p sub-images f^j, $j = 1, ..., p$. The computations proceed as follows, see Fig. 2(b):

- **read:** Every processor reads all the subsets' events.
- **forwproj:** Each processor k_j performs the forward projection for sub-image f^j, i. e., it computes $b_{i,j} = A_i^j f^j$ events $i \in S_l$, where A^j is a sub-matrix of A that is restricted to the voxels in f^j.
- **allreduce:** $b_{i,j}$ are summed up, with the result b_i residing on all processors.
- **update:** Each processor k_j computes $a f^j \sum_{i \in S_l} (A_i^j)^t \frac{1}{b_i}$ for its subimage f^j.



(a) PSD (b) ISD

Fig. 2. Parallel strategies (PSD and ISD) on four distributed-memory processors

On cluster computers, the PSD strategy outperforms the ISD strategy in almost all cases [11], because, in the ISD strategy, large amounts of data have to be read from the remote file system and load imbalances arise from the non-uniform distribution of the radioactive substance inside the reconstruction region. Only in the case of very few events per subset, the ISD strategy is preferable to the PSD strategy.

4 Parallelization: From Distributed-Memory to GPU

4.1 GPU Architecture and Language Support

Modern GPUs (Graphics Processing Units) can be viewed as mathematical coprocessors: they add computing power to the CPU. GPUs are primarily designed for the 3D gaming sector, where they support the CPU of commodity PCs to gain better performance, resulting in high-quality graphics at high-screen resolution.

A GPU is a parallel machine (see Fig. 3) that consists of SIMD (Single Instruction Multiple Data) multiprocessors (ranging from 1 to 16). 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 1.5 GB. On the off-the-shelf main board, one or two GPUs can be installed and used as coprocessors simultaneously. The GeForce 8800 GTX, which we use in our experiments, provides 768 MB device memory and has 16 multiprocessors each with 8 shader units. With CUDA (Compute Unified Device Architecture) [3], the GPU vendor NVIDIA provides a programming interface that introduces the thread-programming concept on 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. Among the main features the CUDA programming interface (present version 1.0) lacks, compared to a traditional thread library like `pthreads`, are mechanisms for mutual

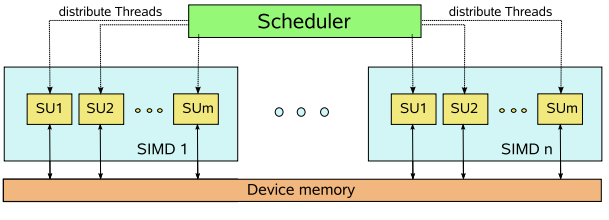


Fig. 3. GPU architecture of modern NVIDIA GPUs: n is the number of multiprocessors, m is the number of shader units. SIMD k denotes the k-th multiprocessor, SU k denotes the k-th shader unit of the multiprocessor.

exclusion like semaphores and monitors. In the next section, we show how the lack of mutual exclusion mechanisms impacts the design and programming of parallel algorithms on the GPU.

4.2 Identification of Data-Parallel Functions Using Petri Nets

Our goal in the Petri Net approach is to systematically develop parallel algorithms for GPUs from HLPN that describe parallel distributed-memory algorithms. HLPN [2] (see Fig. 4 and Fig. 5) consist of places (empty circles), holding data (filled circles), which are linked to transitions (rectangles). If all input places of a transition contain a data element, the transition "fires", i.e., the function corresponding to the transition is executed. The results of the function are placed in one or more output places, which can either be the endpoint of a graph, or again input for preceding transitions. We allow guards for conditional execution of transitions. In order to visualize the parallelism in the Petri Nets, transitions performed on all processors are filled with hatched bars, whereas the fully filled transitions are only computed sequentially on one processor.

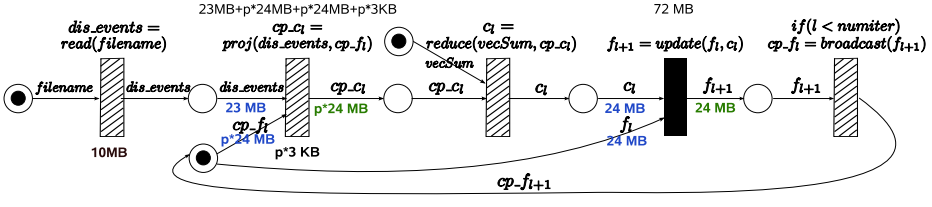


Fig. 4. HLPN for the PSD strategy. The numbers below the different data elements denote their memory requirement.

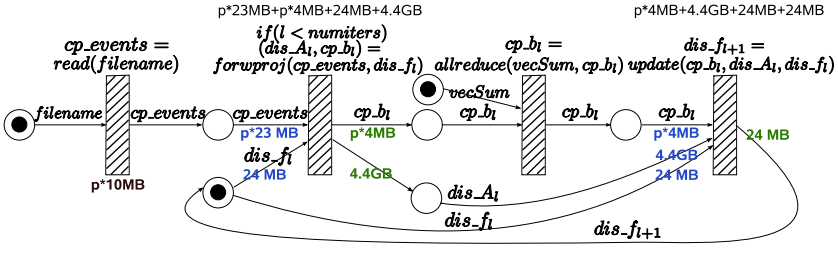


Fig. 5. HLPN for the ISD strategy. The numbers below the different data elements denote their memory requirement.

The idea is to examine two aspects of the parallel algorithm for distributed-memory architectures: 1) the amount of data-parallelism, and 2) the required amount of memory. We start with a HLPN presented in Fig. 4 and Fig. 5. For each of the four steps, we show its application to the parallel PSD and ISD strategy:

1. **Identification of data-parallel functions from parallel transitions:** The HLPN is annotated with additional information: "*dis_*" prefixes are added to the data elements that are distributed among the processors, whereas the data elements with local copies on each processor are denoted with "*cp_*". The "*dis_*" data elements that are input to a parallel transition now indicate data parallelism for the associated parallel function; i.e., the function can be computed simultaneously on different parts of the data. In Fig. 4 and Fig. 5, we annotated the HLPN for the PSD and the ISD strategy with the two prefixes. Since the "*dis_events*" are input to the parallel *proj* function of the PSD strategy and the "*dis_*f_l" are input to the parallel *forwproj* and the parallel *update* function of the ISD strategy, these three functions can be computed in a data-parallel manner.

2. **Identification of data-parallel functions from sequential transitions:** Each sequential function is analyzed for data-parallelism that could not be exploited on the distributed-memory architecture, due to the large communication overhead arising from the distribution and collection of data. In the PSD strategy, the *update* function multiplies f_l and c_l element-wise; this can be done in a data-parallel way.

3. **Annotation of data-parallel transitions with memory requirements:** The data-parallel transitions identified in the two previous steps and their in- and output edges are annotated with their memory requirements. For each data element, the amount of data on the total cluster is annotated; e.g., if one data element takes up k MB on one processor, then the "*cp_*"-prefixed data element takes up p*k MB on the total cluster (p is the number of processors). For each transition, the total memory requirement of the corresponding function is annotated.

4. **Translation and evaluation of memory requirements on the GPU:** The "*dis_*"-annotated data elements will use the same amount of data on the GPU as on the cluster. For "*cp_*" data elements, it has to be decided if the local copies on all processors are identical; if so, only one copy has to be kept in the device memory from which all GPU processors can read and, thus, requirements are divided by p. The data-parallel functions' memory requirements remain unchanged. Now, for each transition, the total amount of required device memory is added up and is compared to the device memory available on a given GPU. As described in steps 1 and 2, we identified four data-parallel functions: the *proj* and *update* functions of the PSD strategy, and the *forwproj* and *update* function of the ISD strategy.

In CUDA, memory latency is hidden by starting several threads per shader unit [8]. Hence, the identified data-parallel functions should provide fine-grained parallelism in order to obtain a fast GPU implementation. However, the "*dis*" prefix with which we identified data-parallel functions (step 1) implies that one thread can be started for each input data element and, thus, fine granularity of the identified functions is guaranteed.

In general, memory requirements are a function in the algorithm's input parameters and input data size. For our particular algorithm, the memory requirements depend on the reconstruction image size and the number of events per subset. In order to simplify matters in this example, we do not determine memory

requirements as a function of parameters, but rather use the particular memory requirements of the reconstruction setup that delivers the most accurate images for our scanner: The image size is $150 \cdot 150 \cdot 280 = 6,300,000$ voxels which take up 24 MB memory and the number of events per subset are 1,000,000 which take up 23 MB memory.

Steps 3 and 4, applied to the *forwproj* function, show that the whole system matrix for each subset $A_l \in \mathbb{R}^{m \text{x} N}$ (m is the number of events per subset), which takes up 4.4 GB memory, has to be kept in the device memory, because it is used again as input to the update function. Thus the memory requirements add up to 24 MB+p*4 MB+4.4 GB≈4.9 GB. With only 768 MB device memory, the ISD strategy is not feasible to be computed on the GPU. The *update* function of the PSD strategy only requires 72 MB device memory (see Fig. 4) and thus can be parallelized on the GPU. The *proj* function also requires 47 MB+p*3 KB+p*24 MB of device memory (see the sum above the *proj* transition in Fig. 4). With the p=128 shader units of our GPU, this adds up to more than 3 GB, with again only 768 MB device memory available. We needed to adapt the original PSD strategy and, therefore, decided not to use separate *proj* and *reduce* steps. Now, each GPU thread writes directly to the shared c_l that resides on the device memory.

One general problem of the CUDA library is its lack of mechanisms for avoiding race conditions (e.g., semaphores). In our case, it would be helpful to protect c_l with a semaphore and thus allow only one thread at a time to write its result. Since this is not supported by CUDA, we decided to allow race conditions due to the following considerations:

- When two threads add one float concurrently to $c_{l,i}$, then, in the worst case, one thread overwrites the result of another; i.e., a small error occurs in $c_{l,i}$.
- The size of c_l is high as compared to the number of parallel writing threads (in our case 6,300,000 voxels vs. 128 threads); therefore, the number of race conditions and thus incorrect voxels is small. We estimated experimentally that only for about 0.04 % of all writes to c_l, a race condition occurs.
- Most importantly: due to the few and very small incorrectnesses in the image, the maximum relative error over all voxels is less than 1 %, which leads to no visual effect on the reconstructed images. However, in order to use the GPU implementation in practice, we will perform a study with many different sets of input data and parameters studied by medical doctors.

With these adjustments to the *proj* function, its memory requirements only add up to 71 MB+p*3 KB≈72 MB for $p = 128$. But even in the best case, where the maximum number of $p = 12288$ threads are started simultaneously, memory requirements only add up to ≈107 MB.

The calculation of one subset on p SIMD processors of the GPU now proceeds as follows:

- **read:** The CPU reads the subsets' events and copies them to the GPU
- **proj:** All processors k_j compute simultaneously $c_{l,j} = \sum_{i \in S_{l,j}} (A_i)^t \frac{1}{A_i f_l}$, $c_{l,j}$ is directly added to c_l.

– **update:** f is divided into sub-images f^j and each processor computes $f_{l+1}^j = f_l^j \, c_l^j$ on its sub-image.

If the target parallel machine has two GPUs available, we have two separate device memories. The computations in this case can proceed as above, with each GPU computing half of the events during the forward-projections and half of the sub-images during the computation of f_{l+1}. After all forward-projections, the two c_ls residing on the device memories are summed up.

The resulting parallel implementation is reusable for different types of PET scanners: a new scanner can be introduced by simply exchanging the sequential projection that computes a_i in the *proj* function. For traditional implementations that parallelize the projection itself, this exchange would require a new, complicated and error-prone parallelization. Furthermore, our implementation is modular, because new functions can be added or interchanged easily; this could be helpful, e.g., when pre- and postprocessing steps are added.

5 Experimental Results (Distributed-Memory vs. GPU)

In our performance experiments, we studied the reconstruction of data collected by the quadHIDAC small-animal PET scanner [10]. We used 10 million events collected during a mouse scan divided in 10 subsets. The reconstruction image has the size $N = (150 \times 150 \times 280)$.

Based on a sequential C++ implementation of the list-mode OSEM algorithm, we used MPI (Message Passing Interface) for our parallelization on a distributed-memory cluster using the two presented decomposition strategies. We performed runtime experiments on the Arminius cluster with 200 Dual INTEL Xeon 3.2 GHZ 64bit nodes, each with 4 GByte main memory, connected by an InfiniBand network. In order to obtain results for the distributed-memory architecture, we used only one of each node's processors. To exploit the fast InfiniBand interconnect, we used the Scali MPI Connect [9] implementation on this machine. In this typical setup, the PSD strategy outperforms the ISD strategy. We measured a minimum runtime of the PSD strategy on 32 processors of

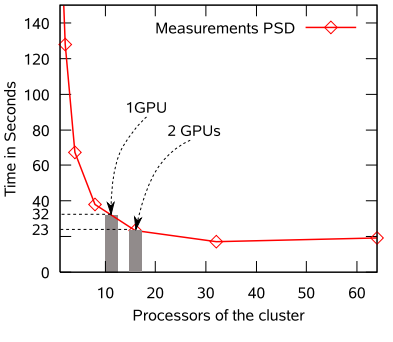


Fig. 6. Runtimes on cluster and GPUs

17.2 seconds (see Fig. 6). With more processors, runtime deteriorates due to the large amounts of data to be communicated per subset. Since sequential runtime measured on one processor of this cluster is about 233 seconds, we achieved a speedup of 13.5 on 32 processors.

For our experiments with graphics processing units, we used two GPUs of the type NVIDIA GeForce 8800 GTX. They have 16 SIMD-multiprocessors, each with 8 shader units running at 1.35 GHz. The device memory is 768 MB. Our CPU is a 2.4 GHz dual-core processor with 2 GB main memory. The average measured runtime is about 32 seconds on one GPU, see Fig. 6. With two GPUs, we achieve a runtime of about 23 seconds. Hence, our parallel GPU parallelization is only about 1.3 times slower than the cluster reconstruction with 32 processors.

We observe similar scalability results for other typical image sizes N on the cluster as well as on the GPU. However, images with $N > 5 \cdot 10^8$ cannot be reconstructed on the GPU because of the small amount of available device memory. This is irrelevant for the quadHIDAC scanner or any other PET scanner at our university hospital, but it could become an issue for future, higher-resolution scanners.

6 Conclusion

We presented a novel approach to systematically develop parallel algorithms for GPUs starting from parallel algorithms for distributed-memory machines. Although we so far limited our considerations to medical imaging, the HLPN approach can be used to analyze data-parallelism and memory requirements for parallel algorithms with the two following properties: First, the distributed-memory implementation has to be made up of separable functions; i.e., each function output is used as input for the subsequent function and no functions are called from within other functions. Second, the memory requirements of the parallel functions can be determined as a function of input parameters and the size of input data. One of the areas where these requirements are fulfilled, are iterative linear system solvers, and, in particular, medical image reconstruction algorithms like the list-mode OSEM algorithm. For the algorithms with data-parallel functions that fulfill the memory requirements, a fast and modular implementation on GPUs can be developed systematically.

For a 3D medical imaging algorithm, we demonstrated in detail how our approach is used to develop a fast, modular and reusable implementation on GPUs. The runtime of the parallel algorithm on two state-of-the-art GPUs is 1.3 times slower than the runtime on 32 processors of a computer cluster. If we take into account the much higher costs for purchase and even more for administration of a distributed-memory cluster, we come to the conclusion that GPU parallelization is a cost-effective choice for list-mode OSEM medical image reconstruction.

Acknowledgments

We thank the NVIDIA corporation for the donation of graphic hardware used in our experiments. We also thank the University of Paderborn for letting us use the

Arminius Cluster. Special thanks go to Dominik Meiländer for his work on the GPU parallelization. This work was partly funded by the Deutsche Forschungsgemeinschaft, SFB 656 MoBil (Project B2). We are grateful to the anonymous referees for their helpful comments.

References

1. Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M., Hanrahan, P.: Brook for GPUs: Stream Computing in graphics hardware. ACM Trans. Graph. 23(3), 777–786 (2004)
2. Girault, C., Valk, R. (eds.): Petri Nets for System Engineers. Springer, Berlin (2003)
3. NVIDIA. NVIDIA CUDA Compute Unified Device Architecture, `http://developer.nvidia.com/object/cuda.html`
4. Natterer, F., Wuebbeling, F.: Mathematical Methods in Image Reconstruction. SIAM, Philadelphia (2001)
5. Owens, J.D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A.E., Purcell, T.J.: A Survey of General-Purpose Computation on Graphics Hardware. Comp. Graph. Forum 26(1), 80–113 (2007)
6. Pratx, G., Chinn, G., Habte, F., Olcott, P., Levin, C.: Fully 3-D list-mode OSEM accelerated by graphics processing units. In: IEEE Nuclear Science Symposium Conference Record, vol. 4, pp. 2196–2202. IEEE, Los Alamitos (2006)
7. Reader, A.J., Erlandsson, K., Flower, M.A., Ott, R.J.: Fast accurate iterative reconstruction for low-statistics positron volume imaging. Phys. Med. Biol. 43(4), 823–834 (1998)
8. Ryoo, S., Rodrigues, C., Baghsorkhi, S., Stone, S., Kirk, D., Hwu, W.: Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In: PPoPP 2008: Proc. of the 13th ACM SIGPLAN Symposium, pp. 73–82 (2008)
9. Scali MPI connect, `http://www.scali.com/`
10. Schäfers, K.P., Reader, A.J., Kriens, M., Knoess, C., Schober, O., Schäfers, M.: Performance evaluation of the 32-module quadHIDAC small-animal PET scanner. Journal Nucl. Med. 46(6), 996–1004 (2005)
11. Schellmann, M., Gorlatch, S.: Comparison of two decomposition strategies for parallelizing the 3D list-mode OSEM algorithm. In: Proceedings Fully 3D Meeting and HPIR Workshop, pp. 37–40 (2007)
12. Xu, F., Mueller, K.: Accelerating popular tomographic reconstruction algorithms on commodity PC graphics hardware. IEEE Trans. Nucl. Sci. 52(3), 654–663 (2005)